

# Intel® 64 and IA-32 Architectures Software Developer's Manual

## Volume 2A: Instruction Set Reference, A-M

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of five volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-M*, Order Number 253666; *Instruction Set Reference N-Z*, Order Number 253667; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669. Refer to all five volumes when evaluating your design needs.

Order Number: 253666-024US
August 2007

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.

Intel may make changes to specifications and product descriptions at any time, without notice.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

The Intel® 64 architecture processors may contain design defects or errors known as errata. Current characterized errata are available on request.

Hyper-Threading Technology requires a computer system with an Intel® processor supporting Hyper-Threading Technology and an HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. For more information, see http://www.intel.com/technology/hyperthread/index.htm; including details on which processors support HT Technology.

No computer system can provide absolute security under all conditions. Intel® Trusted Execution Technology requires a computer system with Intel® Virtualization Technology, an Intel TXT-enabled processor, chipset, BIOS, Authenticated Code Modules and an Intel TXT-compatible measured launched environment (MLE). The MLE could consist of a virtual machine monitor, an OS or an application. In addition, Intel TXT requires the system to contain a TPM v1.2, as defined by the Trusted Computing Group, and specific software for some uses. For more information, see http://www.intel.com/technology/security

Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and for some uses, certain platform software enabled for it. Functionality, performance or other benefits will vary depending on hardware and software configurations. Intel® Virtualization Technology-enabled BIOS and VMM applications are currently in development.

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Processors will not operate (including 32-bit operation) without an Intel® 64 architecture-enabled BIOS. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

Enabling Execute Disable Bit functionality requires a PC with a processor with Execute Disable Bit capability and a supporting operating system. Check with your PC manufacturer on whether your system delivers Execute Disable Bit functionality.

Intel, Pentium, Intel Xeon, Intel NetBurst, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo, Intel Core 2 Extreme, Intel Pentium D, Itanium, Intel SpeedStep, MMX, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 5937
Denver, CO 80217-9808

or call 1-800-548-4725
or visit Intel's website at http://www.intel.com

Copyright © 1997-2007 Intel Corporation

# CONTENTS

PAGE


CHAPTER 1
ABOUT THIS MANUAL
1.1 IA-32 PROCESSORS COVERED IN THIS MANUAL ..... 1-1
1.2 OVERVIEW OF VOLUME 2A AND 2B: INSTRUCTION SET REFERENCE ..... 1-2
1.3 NOTATIONAL CONVENTIONS ..... 1-3
1.3.1 Bit and Byte Order ..... 1-4
1.3.2 Reserved Bits and Software Compatibility ..... 1-5
1.3.3 Instruction Operands ..... 1-6
1.3.4 Hexadecimal and Binary Numbers ..... 1-6
1.3.5 Segmented Addressing ..... 1-6
1.3.6 Exceptions ..... 1-7
1.3.7 A New Syntax for CPUID, CR, and MSR Values ..... 1-7
1.4 RELATED LITERATURE ..... 1-8

CHAPTER 2
INSTRUCTION FORMAT
2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE 2-1
2.1.1 Instruction Prefixes ..... 2-1
2.1.2 Opcodes ..... 2-3
2.1.3 ModR/M and SIB Bytes ..... 2-4
2.1.4 Displacement and Immediate Bytes ..... 2-4
2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes ..... 2-4
2.2 IA-32E MODE ..... 2-9
2.2.1 REX Prefixes ..... 2-9
2.2.1.1 Encoding ..... 2-10
2.2.1.2 More on REX Prefix Fields ..... 2-10
2.2.1.3 Displacement ..... 2-13
2.2.1.4 Direct Memory-Offset MOVs ..... 2-13
2.2.1.5 Immediates ..... 2-14
2.2.1.6 RIP-Relative Addressing ..... 2-14
2.2.1.7 Default 64-Bit Operand Size ..... 2-15
2.2.2 Additional Encodings for Control and Debug Registers ..... 2-15

CHAPTER 3
INSTRUCTION SET REFERENCE, A-M
3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES ..... 3-1
3.1.1 Instruction Format ..... 3-1
3.1.1.1 Opcode Column in the Instruction Summary Table ..... 3-2
3.1.1.2 Instruction Column in the Opcode Summary Table ..... 3-3
3.1.1.3 64-bit Mode Column in the Instruction Summary Table ..... 3-6
3.1.1.4 Compatibility/Legacy Mode Column in the Instruction Summary Table ..... 3-7

PAGE

3.1.1.5 Description Column in the Instruction Summary Table . . . 3-7
3.1.1.6 Description Section . . . 3-7
3.1.1.7 Operation Section . . . 3-7
3.1.1.8 Intel® C/C++ Compiler Intrinsics Equivalents Section . . . 3-11
3.1.1.9 Flags Affected Section . . . 3-14
3.1.1.10 FPU Flags Affected Section . . . 3-14
3.1.1.11 Protected Mode Exceptions Section . . . 3-14
3.1.1.12 Real-Address Mode Exceptions Section . . . 3-16
3.1.1.13 Virtual-8086 Mode Exceptions Section . . . 3-16
3.1.1.14 Floating-Point Exceptions Section . . . 3-16
3.1.1.15 SIMD Floating-Point Exceptions Section . . . 3-17
3.1.1.16 Compatibility Mode Exceptions Section . . . 3-17
3.1.1.17 64-Bit Mode Exceptions Section . . . 3-17
3.2 INSTRUCTIONS (A-M) . . . 3-18
AAA—ASCII Adjust After Addition . . . 3-19
AAD—ASCII Adjust AX Before Division . . . 3-21
AAM—ASCII Adjust AX After Multiply . . . 3-23
AAS—ASCII Adjust AL After Subtraction . . . 3-25
ADC—Add with Carry . . . 3-27
ADD—Add . . . 3-30
ADDPD—Add Packed Double-Precision Floating-Point Values . . . 3-33
ADDPS—Add Packed Single-Precision Floating-Point Values . . . 3-36
ADDSD—Add Scalar Double-Precision Floating-Point Values . . . 3-39
ADDSS—Add Scalar Single-Precision Floating-Point Values . . . 3-42
ADDSUBPD—Packed Double-FP Add/Subtract . . . 3-45
ADDSUBPS—Packed Single-FP Add/Subtract . . . 3-49
AND—Logical AND . . . 3-53
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values . . . 3-56
ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values . . . 3-58
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values . . . 3-60
ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values . . . 3-62
ARPL—Adjust RPL Field of Segment Selector . . . 3-64
BOUND—Check Array Index Against Bounds . . . 3-66
BSF—Bit Scan Forward . . . 3-69
BSR—Bit Scan Reverse . . . 3-71
BSWAP—Byte Swap . . . 3-73
BT—Bit Test . . . 3-75
BTC—Bit Test and Complement . . . 3-78
BTR—Bit Test and Reset . . . 3-81
BTS—Bit Test and Set . . . 3-84
CALL—Call Procedure . . . 3-87
CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword . . . 3-105
CLC—Clear Carry Flag . . . 3-106
CLD—Clear Direction Flag . . . 3-107

PAGE

CLFLUSH—Flush Cache Line . . . . . 3-108
CLI — Clear Interrupt Flag . . . . . 3-110
CLTS—Clear Task-Switched Flag in CR0 . . . . . 3-113
CMC—Complement Carry Flag . . . . . 3-115
CMOVcc—Conditional Move . . . . . 3-116
CMP—Compare Two Operands . . . . . 3-123
CMPPD—Compare Packed Double-Precision Floating-Point Values . . . . . 3-126
CMPPS—Compare Packed Single-Precision Floating-Point Values . . . . . 3-131
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands . . . . . 3-136
CMPSD—Compare Scalar Double-Precision Floating-Point Values . . . . . 3-142
CMPSS—Compare Scalar Single-Precision Floating-Point Values . . . . . 3-146
CMPXCHG—Compare and Exchange . . . . . 3-150
CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes . . . . . 3-153
COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS . . . . . 3-156
COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS . . . . . 3-159
CPUID—CPU Identification . . . . . 3-162
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values . . . . . 3-190
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values . . . . . 3-192
CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers . . . . . 3-195
CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers . . . . . 3-198
CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values . . . . . 3-201
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values . . . . . 3-204
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values . . . . . 3-207
CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers . . . . . 3-210
CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values . . . . . 3-213
CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers . . . . . 3-216
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer . . . . . 3-219
CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value . . . . . 3-222
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value . . . . . 3-225
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value . . . . . 3-228
CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar

PAGE

Double-Precision Floating-Point Value....3-231
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer....3-234
CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers....3-237
CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers....3-240
CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers....3-243
CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers....3-246
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer....3-249
CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer....3-252
CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword3-255
DAA—Decimal Adjust AL after Addition....3-257
DAS—Decimal Adjust AL after Subtraction....3-259
DEC—Decrement by 1....3-261
DIV—Unsigned Divide....3-264
DIVPD—Divide Packed Double-Precision Floating-Point Values....3-268
DIVPS—Divide Packed Single-Precision Floating-Point Values....3-271
DIVSD—Divide Scalar Double-Precision Floating-Point Values....3-274
DIVSS—Divide Scalar Single-Precision Floating-Point Values....3-277
EMMS—Empty MMX Technology State....3-280
ENTER—Make Stack Frame for Procedure Parameters....3-282
F2XM1—Compute 2x–1....3-286
FABS—Absolute Value....3-288
FADD/FADDP/FIADD—Add....3-290
FBLD—Load Binary Coded Decimal....3-294
FBSTP—Store BCD Integer and Pop....3-296
FCHS—Change Sign....3-299
FCLEX/FNCLEX—Clear Exceptions....3-301
FCMOVcc—Floating-Point Conditional Move....3-303
FCOMI/FCOMIP/ FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS....3-309
FCOS—Cosine....3-312
FDECSTP—Decrement Stack-Top Pointer....3-314
FDIV/FDIVP/FIDIV—Divide....3-316
FDIVR/FDIVRP/FIDIVR—Reverse Divide....3-320
FFREE—Free Floating-Point Register....3-324
FICOM/FICOMP—Compare Integer....3-325
FILD—Load Integer....3-328
FINCSTP—Increment Stack-Top Pointer....3-330
FINIT/FNINIT—Initialize Floating-Point Unit....3-332
FIST/FISTP—Store Integer....3-334
FISTTP—Store Integer with Truncation....3-338

PAGE

FLD—Load Floating Point Value ..... 3-341
FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant ..... 3-344
FLDCW—Load x87 FPU Control Word ..... 3-346
FLDENV—Load x87 FPU Environment ..... 3-348
FMUL/FMULP/FIMUL—Multiply ..... 3-351
FNOP—No Operation ..... 3-355
FPATAN—Partial Arctangent ..... 3-356
FPREM—Partial Remainder ..... 3-359
FPREM1—Partial Remainder ..... 3-362
FPTAN—Partial Tangent ..... 3-365
FRNDINT—Round to Integer ..... 3-368
FRSTOR—Restore x87 FPU State ..... 3-370
FSAVE/FNSAVE—Store x87 FPU State ..... 3-373
FSCALE—Scale ..... 3-377
FSIN—Sine ..... 3-379
FSINCOS—Sine and Cosine ..... 3-381
FSQRT—Square Root ..... 3-384
FST/FSTP—Store Floating Point Value ..... 3-386
FSTCW/FNSTCW—Store x87 FPU Control Word ..... 3-389
FSTENV/FNSTENV—Store x87 FPU Environment ..... 3-392
FSTSW/FNSTSW—Store x87 FPU Status Word ..... 3-395
FSUB/FSUBP/FISUB—Subtract ..... 3-398
FSUBR/FSUBRP/FISUBR—Reverse Subtract ..... 3-402
FTST—TEST ..... 3-406
FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values ..... 3-408
FXAM—ExamineModR/M ..... 3-411
FXCH—Exchange Register Contents ..... 3-413
FXRSTOR—Restore x87 FPU, MMX , XMM, and MXCSR State ..... 3-415
FXSAVE—Save x87 FPU, MMX Technology, SSE, and SSE2 State ..... 3-418
FXTRACT—Extract Exponent and Significand ..... 3-429
FYL2X—Compute y * log2x ..... 3-431
FYL2XP1—Compute y * log2(x +1) ..... 3-433
HADDPD—Packed Double-FP Horizontal Add ..... 3-435
HADDPS—Packed Single-FP Horizontal Add ..... 3-439
HLT—Halt ..... 3-443
HSUBPD—Packed Double-FP Horizontal Subtract ..... 3-445
HSUBPS—Packed Single-FP Horizontal Subtract ..... 3-449
IDIV—Signed Divide ..... 3-453
IMUL—Signed Multiply ..... 3-457
IN—Input from Port ..... 3-462
INC—Increment by 1 ..... 3-464
INS/INSB/INSW/INSD—Input from Port to String ..... 3-467
INT n/INTO/INT 3—Call to Interrupt Procedure ..... 3-471
INVD—Invalidate Internal Caches ..... 3-486
INVLPG—Invalidate TLB Entry ..... 3-488
IRET/IRETD—Interrupt Return ..... 3-490
Jcc—Jump if Condition Is Met ..... 3-501

PAGE

JMP—Jump ..... 3-508
LAHF—Load Status Flags into AH Register ..... 3-518
LAR—Load Access Rights Byte. ..... 3-520
LDDQU—Load Unaligned Integer 128 Bits ..... 3-524
LDMXCSR—Load MXCSR Register ..... 3-527
LDS/LES/LFS/LGS/LSS—Load Far Pointer ..... 3-529
LEA—Load Effective Address ..... 3-535
LEAVE—High Level Procedure Exit ..... 3-538
LFENCE—Load Fence ..... 3-540
LGDT/LIDT—Load Global/Interrupt Descriptor Table Register ..... 3-541
LLDT—Load Local Descriptor Table Register ..... 3-544
LMSW—Load Machine Status Word ..... 3-547
LOCK—Assert LOCK# Signal Prefix ..... 3-549
LODS/LODSB/LODSW/LODSD/LODSQ—Load String ..... 3-551
LOOP/LOOPcc—Loop According to ECX Counter ..... 3-555
LSL—Load Segment Limit ..... 3-558
LTR—Load Task Register ..... 3-562
MASKMOVDQU—Store Selected Bytes of Double Quadword ..... 3-565
MASKMOVQ—Store Selected Bytes of Quadword ..... 3-568
MAXPD—Return Maximum Packed Double-Precision Floating-Point Values ..... 3-571
MAXPS—Return Maximum Packed Single-Precision Floating-Point Values ..... 3-574
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value ..... 3-577
MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value ..... 3-580
MFENCE—Memory Fence ..... 3-583
MINPD—Return Minimum Packed Double-Precision Floating-Point Values ..... 3-584
MINPS—Return Minimum Packed Single-Precision Floating-Point Values ..... 3-587
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value ..... 3-590
MINSS—Return Minimum Scalar Single-Precision Floating-Point Value ..... 3-593
MONITOR—Set Up Monitor Address ..... 3-596
MOV—Move ..... 3-599
MOV—Move to/from Control Registers ..... 3-605
MOV—Move to/from Debug Registers ..... 3-608
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values ..... 3-610
MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values ..... 3-613
MOVD/MOVQ—Move Doubleword/Move Quadword ..... 3-616
MOVDDUP—Move One Double-FP and Duplicate ..... 3-620
MOVDQA—Move Aligned Double Quadword. ..... 3-623
MOVDQU—Move Unaligned Double Quadword ..... 3-625
MOVDQ2Q—Move Quadword from XMM to MMX Technology Register ..... 3-628
MOVHLPS— Move Packed Single-Precision Floating-Point Values High to Low ..... 3-630
MOVHPD—Move High Packed Double-Precision Floating-Point Value ..... 3-632
MOVHPS—Move High Packed Single-Precision Floating-Point Values ..... 3-635
MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High. ..... 3-638
MOVLPD—Move Low Packed Double-Precision Floating-Point Value. ..... 3-640
MOVLPS—Move Low Packed Single-Precision Floating-Point Values. ..... 3-642
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask ..... 3-645
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask ..... 3-647

MOVNTDQ—Store Double Quadword Using Non-Temporal Hint ..... 3-649
MOVNTI—Store Doubleword Using Non-Temporal Hint ..... 3-652
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint ..... 3-654
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint ..... 3-657
MOVNTQ—Store of Quadword Using Non-Temporal Hint ..... 3-660
MOVQ—Move Quadword ..... 3-663
MOVQ2DQ—Move Quadword from MMX Technology to XMM Register ..... 3-666
MOVS/MOVSB/MOVSW/MOVSD/MOVSQ—Move Data from String to String ..... 3-668
MOVSD—Move Scalar Double-Precision Floating-Point Value ..... 3-673
MOVSHDUP—Move Packed Single-FP High and Duplicate ..... 3-676
MOVSLDUP—Move Packed Single-FP Low and Duplicate ..... 3-679
MOVSS—Move Scalar Single-Precision Floating-Point Values ..... 3-682
MOVSX/MOVSXD—Move with Sign-Extension ..... 3-685
MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values ..... 3-687
MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values ..... 3-690
MOVZX—Move with Zero-Extend ..... 3-693
MUL—Unsigned Multiply ..... 3-695
MULPD—Multiply Packed Double-Precision Floating-Point Values ..... 3-698
MULPS—Multiply Packed Single-Precision Floating-Point Values ..... 3-701
MULSD—Multiply Scalar Double-Precision Floating-Point Values ..... 3-704
MULSS—Multiply Scalar Single-Precision Floating-Point Values ..... 3-707
MWAIT—Monitor Wait ..... 3-710

CHAPTER 4
INSTRUCTION SET REFERENCE, N-Z

4.1 INSTRUCTIONS (N-Z) ..... 4-1
NEG—Two's Complement Negation ..... 4-2
NOP—No Operation ..... 4-5
NOT—One's Complement Negation ..... 4-7
OR—Logical Inclusive OR ..... 4-9
ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values ..... 4-12
ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values ..... 4-14
OUT—Output to Port ..... 4-16
OUTS/OUTSB/OUTSW/OUTSD—Output String to Port ..... 4-18
PABSB/PABSW/PABSD — Packed Absolute Value ..... 4-23
PACKSSWB/PACKSSDW—Pack with Signed Saturation ..... 4-27
PACKUSWB—Pack with Unsigned Saturation ..... 4-32
PADDB/PADDW/PADDD—Add Packed Integers ..... 4-36
PADDQ—Add Packed Quadword Integers ..... 4-40
PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation ..... 4-43
PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation ..... 4-47
PALIGNR — Packed Align Right ..... 4-51
PAND—Logical AND ..... 4-54
PANDN—Logical AND NOT ..... 4-57
PAUSE—Spin Loop Hint ..... 4-60

PAGE

PAVGB/PAVGW—Average Packed Integers . . . . . 4-61
PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal. . . . . 4-64
PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than . 4-68
PEXTRW—Extract Word . . . . . 4-73
PHADDW/PHADDD — Packed Horizontal Add . . . . . 4-76
PHADDSW — Packed Horizontal Add and Saturate . . . . . 4-79
PHSUBW/PHSUBD — Packed Horizontal Subtract . . . . . 4-82
PHSUBSW — Packed Horizontal Subtract and Saturate. . . . . 4-85
PINSRW—Insert Word . . . . . 4-88
PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes. . . . . 4-91
PMADDWD—Multiply and Add Packed Integers. . . . . 4-94
PMAXSW—Maximum of Packed Signed Word Integers . . . . . 4-98
PMAXUB—Maximum of Packed Unsigned Byte Integers . . . . . 4-101
PMINSW—Minimum of Packed Signed Word Integers . . . . . 4-104
PMINUB—Minimum of Packed Unsigned Byte Integers . . . . . 4-107
PMOVMSKB—Move Byte Mask . . . . . 4-110
PMULHRSW — Packed Multiply High with Round and Scale . . . . . 4-113
PMULHUW—Multiply Packed Unsigned Integers and Store High Result . . . . . 4-116
PMULHW—Multiply Packed Signed Integers and Store High Result . . . . . 4-120
PMULLW—Multiply Packed Signed Integers and Store Low Result. . . . . 4-123
PMULUDQ—Multiply Packed Unsigned Doubleword Integers . . . . . 4-127
POP—Pop a Value from the Stack . . . . . 4-130
POPA/POPAD—Pop All General-Purpose Registers . . . . . 4-137
POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register . . . . . 4-139
POR—Bitwise Logical OR . . . . . 4-143
PREFETCHh—Prefetch Data Into Caches . . . . . 4-146
PSADBW—Compute Sum of Absolute Differences. . . . . 4-148
PSHUFB — Packed Shuffle Bytes . . . . . 4-152
PSHUFD—Shuffle Packed Doublewords . . . . . 4-156
PSHUFHW—Shuffle Packed High Words . . . . . 4-159
PSHUFLW—Shuffle Packed Low Words. . . . . 4-162
PSHUFW—Shuffle Packed Words. . . . . 4-165
PSIGNB/PSIGNW/PSIGND — Packed SIGN . . . . . 4-168
PSLLDQ—Shift Double Quadword Left Logical . . . . . 4-173
PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical. . . . . 4-175
PSRAW/PSRAD—Shift Packed Data Right Arithmetic . . . . . 4-180
PSRLDQ—Shift Double Quadword Right Logical . . . . . 4-185
PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical. . . . . 4-187
PSUBB/PSUBW/PSUBD—Subtract Packed Integers. . . . . 4-192
PSUBQ—Subtract Packed Quadword Integers. . . . . 4-196
PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation . . . . . 4-199
PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation . . . . . 4-203
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data . . . . 4-207
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ— Unpack Low Data . . . . . 4-212

PAGE

PUSH—Push Word, Doubleword or Quadword Onto the Stack . . . . . 4-217
PUSHA/PUSHAD—Push All General-Purpose Registers. . . . . 4-222
PUSHF/PUSHFD—Push EFLAGS Register onto the Stack . . . . . 4-225
PXOR—Logical Exclusive OR. . . . . 4-228
RCL/RCR/ROL/ROR-—Rotate . . . . . 4-231
RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values . . 4-238
RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values . . . . 4-241
RDMSR—Read from Model Specific Register. . . . . 4-244
RDPMC—Read Performance-Monitoring Counters . . . . . 4-246
RDTSC—Read Time-Stamp Counter . . . . . 4-251
REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix . . . . . 4-253
RET—Return from Procedure . . . . . 4-258
RSM—Resume from System Management Mode. . . . . 4-270
RSQRTPS—Compute Reciprocals of Square Roots of Packed
Single-Precision Floating-Point Values . . . . . 4-272
RSQRTSS—Compute Reciprocal of Square Root of Scalar
Single-Precision Floating-Point Value . . . . . 4-275
SAHF—Store AH into Flags. . . . . 4-278
SAL/SAR/SHL/SHR—Shift . . . . . 4-280
SBB—Integer Subtraction with Borrow . . . . . 4-287
SCAS/SCASB/SCASW/SCASD—Scan String . . . . . 4-291
SETcc—Set Byte on Condition . . . . . 4-296
SFENCE—Store Fence. . . . . 4-301
SGDT—Store Global Descriptor Table Register. . . . . 4-302
SHLD—Double Precision Shift Left. . . . . 4-305
SHRD—Double Precision Shift Right . . . . . 4-308
SHUFPD—Shuffle Packed Double-Precision Floating-Point Values . . . . . 4-311
SHUFPS—Shuffle Packed Single-Precision Floating-Point Values . . . . . 4-314
SIDT—Store Interrupt Descriptor Table Register . . . . . 4-317
SLDT—Store Local Descriptor Table Register. . . . . 4-320
SMSW—Store Machine Status Word . . . . . 4-322
SQRTPS—Compute Square Roots of Packed Single-Precision
Floating-Point Values . . . . . 4-328
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value. 4-331
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value. . 4-334
STC—Set Carry Flag . . . . . 4-337
STD—Set Direction Flag. . . . . 4-338
STI—Set Interrupt Flag. . . . . 4-339
STMXCSR—Store MXCSR Register State . . . . . 4-342
STOS/STOSB/STOSW/STOSD/STOSQ—Store String. . . . . 4-344
STR—Store Task Register. . . . . 4-348
SUB—Subtract . . . . . 4-350
SUBPD—Subtract Packed Double-Precision Floating-Point Values . . . . . 4-353
SUBPS—Subtract Packed Single-Precision Floating-Point Values . . . . . 4-356
SUBSD—Subtract Scalar Double-Precision Floating-Point Values . . . . . 4-359
SUBSS—Subtract Scalar Single-Precision Floating-Point Values. . . . . 4-362
SWAPGS—Swap GS Base Register . . . . . 4-365

PAGE

SYSCALL—Fast System Call....4-367
SYSENTER—Fast System Call....4-369
SYSEXIT—Fast Return from Fast System Call....4-373
SYSRET—Return From Fast System Call....4-377
TEST—Logical Compare....4-379
UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS....4-382
UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS....4-385
UD2—Undefined Instruction....4-388
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values....4-389
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values....4-392
UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values....4-395
UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values....4-398
VERR/VERW—Verify a Segment for Reading or Writing....4-401
WAIT/FWAIT—Wait....4-404
WBINVD—Write Back and Invalidate Cache....4-406
WRMSR—Write to Model Specific Register....4-408
XADD—Exchange and Add....4-410
XCHG—Exchange Register/Memory with Register....4-413
XLAT/XLATB—Table Look-up Translation....4-416
XOR—Logical Exclusive OR....4-418
XORPD—Bitwise Logical XOR for Double-Precision Floating-Point Values....4-421
XORPS—Bitwise Logical XOR for Single-Precision Floating-Point Values....4-423

CHAPTER 5
VMX INSTRUCTION REFERENCE

5.1 OVERVIEW....5-1
5.2 CONVENTIONS....5-2
5.3 VMX INSTRUCTIONS....5-3
VMCALL—Call to VM Monitor....5-4
VMCLEAR—Clear Virtual-Machine Control Structure....5-7
VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine....5-10
VMPTRLD—Load Pointer to Virtual-Machine Control Structure....5-13
VMPTRST—Store Pointer to Virtual-Machine Control Structure....5-16
VMREAD—Read Field from Virtual-Machine Control Structure....5-18
VMRESUME—Resume Virtual Machine....5-21
VMWRITE—Write Field to Virtual-Machine Control Structure....5-22
VMXOFF—Leave VMX Operation....5-25
VMXON—Enter VMX Operation....5-27

## CHAPTER 6
## SAFER MODE EXTENSIONS REFERENCE

6.1 OVERVIEW ..... 5-1
6.2 SMX FUNCTIONALITY ..... 5-1
6.2.1 Detecting and Enabling SMX ..... 5-2
6.2.2 SMX Instruction Summary ..... 5-3
6.2.2.1 GETSEC[CAPABILITIES] ..... 5-3
6.2.2.2 GETSEC[ENTERACCS] ..... 5-4
6.2.2.3 GETSEC[EXITAC] ..... 5-4
6.2.2.4 GETSEC[SENTER] ..... 5-4
6.2.2.5 GETSEC[SEXIT] ..... 5-5
6.2.2.6 GETSEC[PARAMETERS] ..... 5-5
6.2.2.7 GETSEC[SMCTRL] ..... 5-6
6.2.2.8 GETSEC[WAKEUP] ..... 5-6
6.2.3 Measured Environment and SMX ..... 5-6
6.3 GETSEC LEAF FUNCTIONS ..... 5-7
GETSEC[CAPABILITIES] - Report the SMX Capabilities ..... 5-9
GETSEC[ENTERACCS] - Execute Authenticated Chipset Code ..... 5-12
GETSEC[EXITAC]—Exit Authenticated Code Execution Mode ..... 5-23
GETSEC[SENTER]—Enter a measured environment ..... 5-27
GETSEC[SEXIT]—Exit measured environment ..... 5-39
GETSEC[PARAMETERS]—Report the SMX parameters ..... 5-43
GETSEC[SMCTRL]—SMX mode control ..... 5-48
GETSEC[WAKEUP]—Wake up sleeping processors in measured environment ..... 5-52

## APPENDIX A
## OPCODE MAP

A.1 USING OPCODE TABLES ..... A-1
A.2 KEY TO ABBREVIATIONS ..... A-2
A.2.1 Codes for Addressing Method ..... A-2
A.2.2 Codes for Operand Type ..... A-3
A.2.3 Register Codes ..... A-4
A.2.4 Opcode Look-up Examples for One, Two, and Three-Byte OpcodesA-4
A.2.4.1 One-Byte Opcode Instructions ..... A-4
A.2.4.2 Two-Byte Opcode Instructions ..... A-5
A.2.4.3 Three-Byte Opcode Instructions ..... A-6
A.2.5 Superscripts Utilized in Opcode Tables ..... A-7
A.3 ONE, TWO, AND THREE-BYTE OPCODE MAPS ..... A-8
A.4 OPCODE EXTENSIONS FOR ONE-BYTE AND TWO-BYTE OPCODES ..... A-19
A.4.1 Opcode Look-up Examples Using Opcode Extensions ..... A-19
A.4.2 Opcode Extension Tables ..... A-20
A.5 ESCAPE OPCODE INSTRUCTIONS ..... A-22
A.5.1 Opcode Look-up Examples for Escape Instruction Opcodes ..... A-22
A.5.2 Escape Opcode Instruction Tables ..... A-22
A.5.2.1 Escape Opcodes with D8 as First Byte ..... A-23
A.5.2.2 Escape Opcodes with D9 as First Byte ..... A-24

PAGE

A.5.2.3 Escape Opcodes with DA as First Byte ..... A-25
A.5.2.4 Escape Opcodes with DB as First Byte ..... A-26
A.5.2.5 Escape Opcodes with DC as First Byte..... A-27
A.5.2.6 Escape Opcodes with DD as First Byte ..... A-28
A.5.2.7 Escape Opcodes with DE as First Byte..... A-29
A.5.2.8 Escape Opcodes with DF As First Byte ..... A-30

APPENDIX B
INSTRUCTION FORMATS AND ENCODINGS

B.1 MACHINE INSTRUCTION FORMAT ..... B-1
B.1.1 Legacy Prefixes..... B-2
B.1.2 REX Prefixes ..... B-2
B.1.3 Opcode Fields..... B-2
B.1.4 Special Fields ..... B-2
B.1.4.1 Reg Field (reg) for Non-64-Bit Modes ..... B-3
B.1.4.2 Reg Field (reg) for 64-Bit Mode ..... B-4
B.1.4.3 Encoding of Operand Size (w) Bit ..... B-5
B.1.4.4 Sign-Extend (s) Bit ..... B-5
B.1.4.5 Segment Register (sreg) Field ..... B-6
B.1.4.6 Special-Purpose Register (eee) Field ..... B-6
B.1.4.7 Condition Test (tttn) Field ..... B-7
B.1.4.8 Direction (d) Bit ..... B-8
B.1.5 Other Notes ..... B-9
B.2 GENERAL-PURPOSE INSTRUCTION FORMATS AND ENCODINGS FOR NON-64-BIT MODES ..... B-9
B.2.1 General Purpose Instruction Formats and Encodings for 64-Bit Mode ..... B-24
B.3 PENTIUM® PROCESSOR FAMILY INSTRUCTION FORMATS AND ENCODINGS ..... B-53
B.4 64-BIT MODE INSTRUCTION ENCODINGS FOR SIMD INSTRUCTION EXTENSIONS ..... B-54
B.5 MMX INSTRUCTION FORMATS AND ENCODINGS ..... B-54
B.5.1 Granularity Field (gg)..... B-54
B.5.2 MMX Technology and General-Purpose Register Fields (mmxreg and reg) ..... B-55
B.5.3 MMX Instruction Formats and Encodings Table ..... B-55
B.6 P6 FAMILY INSTRUCTION FORMATS AND ENCODINGS..... B-58
B.7 SSE INSTRUCTION FORMATS AND ENCODINGS..... B-59
B.8 SSE2 INSTRUCTION FORMATS AND ENCODINGS ..... B-68
B.8.1 Granularity Field (gg)..... B-68
B.9 SSE3 FORMATS AND ENCODINGS TABLE ..... B-85
B.10 SSSE3 FORMATS AND ENCODING TABLE ..... B-87
B.11 SPECIAL ENCODINGS FOR 64-BIT MODE ..... B-91
B.12 FLOATING-POINT INSTRUCTION FORMATS AND ENCODINGS ..... B-95
B.13 VMX INSTRUCTIONS ..... B-101
B.14 SMX INSTRUCTIONS..... B-103

APPENDIX C
INTEL® C/C++ COMPILER INTRINSICS AND FUNCTIONAL EQUIVALENTS

C.1 SIMPLE INTRINSICS..... C-2
C.2 COMPOSITE INTRINSICS ..... C-14

# FIGURES

Figure 1-1. Bit and Byte Order ..... 1-5
Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation. ..... 1-8
Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format ..... 2-1
Figure 2-2. Table Interpretation of ModR/M Byte (C8H) ..... 2-5
Figure 2-3. Prefix Ordering in 64-bit Mode ..... 2-9
Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used ..... 2-11
Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used ..... 2-11
Figure 2-6. Memory Addressing With a SIB Byte ..... 2-12
Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used ..... 2-12
Figure 3-1. Bit Offset for BIT[RAX, 21] ..... 3-10
Figure 3-2. Memory Bit Indexing. ..... 3-11
Figure 3-3. ADDSUBPD—Packed Double-FP Add/Subtract ..... 3-45
Figure 3-4. ADDSUBPS—Packed Single-FP Add/Subtract ..... 3-49
Figure 3-5. Version Information Returned by CPUID in EAX ..... 3-170
Figure 3-6. Feature Information Returned in the ECX Register ..... 3-172
Figure 3-7. Feature Information Returned in the EDX Register ..... 3-174
Figure 3-8. Determination of Support for the Processor Brand String. ..... 3-182
Figure 3-9. Algorithm for Extracting Maximum Processor Frequency ..... 3-184
Figure 3-10. HADDPD—Packed Double-FP Horizontal Add ..... 3-435
Figure 3-11. HADDPS—Packed Single-FP Horizontal Add ..... 3-439
Figure 3-12. HSUBPD—Packed Double-FP Horizontal Subtract. ..... 3-445
Figure 3-13. HSUBPS—Packed Single-FP Horizontal Subtract ..... 3-450
Figure 3-14. MOVDDUP—Move One Double-FP and Duplicate ..... 3-620
Figure 3-15. MOVSHDUP—Move Packed Single-FP High and Duplicate ..... 3-676
Figure 3-16. MOVSLDUP—Move Packed Single-FP Low and Duplicate ..... 3-679
Figure 4-1. Operation of the PACKSSDW Instruction Using 64-bit Operands. ..... 4-27
Figure 4-2. PMADDWD Execution Model Using 64-bit Operands ..... 4-95
Figure 4-3. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands ..... 4-116
Figure 4-4. PMULLU Instruction Operation Using 64-bit Operands ..... 4-123
Figure 4-5. PSADBW Instruction Operation Using 64-bit Operands. ..... 4-149
Figure 4-6. PSHUB with 64-Bit Operands ..... 4-153
Figure 4-7. PSHUFD Instruction Operation ..... 4-156
Figure 4-8. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand ..... 4-176
Figure 4-9. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand ..... 4-181
Figure 4-10. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand ..... 4-188
Figure 4-11. PUNPCKHBW Instruction Operation Using 64-bit Operands ..... 4-208
Figure 4-12. PUNPCKLBW Instruction Operation Using 64-bit Operands. ..... 4-212
Figure 4-13. SHUFPD Shuffle Operation ..... 4-311
Figure 4-14. SHUFPS Shuffle Operation ..... 4-314
Figure 4-15. UNPCKHPD Instruction High Unpack and Interleave Operation ..... 4-389
Figure 4-16. UNPCKHPS Instruction High Unpack and Interleave Operation ..... 4-392
Figure 4-17. UNPCKLPD Instruction Low Unpack and Interleave Operation ..... 4-395
Figure 4-18. UNPCKLPS Instruction Low Unpack and Interleave Operation ..... 4-398
Figure A-1. ModR/M Byte nnn Field (Bits 5, 4, and 3) ..... A-19
Figure B-1. General Machine Instruction Format ..... B-1

PAGE

# TABLES


Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte. 2-6
Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte. 2-7
Table 2-3. 32-Bit Addressing Forms with the SIB Byte 2-8
Table 2-4. REX Prefix Fields [BITS: 0100WRXB]. 2-11
Table 2-5. Special Cases of REX Encodings. 2-13
Table 2-6. Direct Memory Offset Form of MOV. 2-14
Table 2-7. RIP-Relative Addressing. 2-15
Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro. 3-2
Table 3-2. Range of Bit Positions Specified by Bit Offset Operands 3-11
Table 3-3. Intel 64 and IA-32 General Exceptions 3-15
Table 3-5. SIMD Floating-Point Exceptions 3-17
Table 3-4. x87 FPU Floating-Point Exceptions 3-17
Table 3-6. Decision Table for CLI Results 3-110
Table 3-7. Comparison Predicate for CMPPD and CMPPS Instructions 3-126
Table 3-8. Pseudo-Op and CMPPD Implementation 3-127
Table 3-9. Pseudo-Ops and CMPPS 3-132
Table 3-10. Pseudo-Ops and CMPSD 3-142
Table 3-11. Pseudo-Ops and CMPSS 3-147
Table 3-12. Information Returned by CPUID Instruction. 3-163
Table 3-13. Highest CPUID Source Operand for Intel 64 and IA-32 Processors 3-169
Table 3-14. Processor Type Field 3-170
Table 3-15. Feature Information Returned in the ECX Register. 3-173
Table 3-16. More on Feature Information Returned in the EDX Register 3-175
Table 3-17. Encoding of Cache and TLB Descriptors 3-178
Table 3-18. Processor Brand String Returned with Pentium 4 Processor. 3-183
Table 3-19. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings. 3-185
Table 3-20. DIV Action 3-265
Table 3-21. Results Obtained from F2XM1 3-286
Table 3-22. Results Obtained from FABS. 3-288
Table 3-23. FADD/FADDP/FIADD Results. 3-291
Table 3-24. FBSTP Results 3-296
Table 3-25. FCHS Results 3-299
Table 3-26. FCOM/FCOMP/FCOMPP Results 3-305
Table 3-27. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results. 3-309
Table 3-28. FCOS Results 3-312
Table 3-29. FDIV/FDIVP/FIDIV Results 3-317
Table 3-30. FDIVR/FDIVRP/FIDIVR Results 3-321
Table 3-31. FICOM/FICOMP Results 3-325
Table 3-32. FIST/FISTP Results. 3-334
Table 3-33. FISTTP Results 3-338
Table 3-34. FMUL/FMULP/FIMUL Results 3-352
Table 3-35. FPATAN Results 3-357
Table 3-36. FPREM Results. 3-359
Table 3-37. FPREM1 Results 3-362

PAGE

Table 3-38. FPTAN Results . . . 3-365
Table 3-39. FSCALE Results . . . 3-377
Table 3-40. FSIN Results . . . 3-379
Table 3-41. FSINCOS Results . . . 3-381
Table 3-42. FSQRT Results . . . 3-384
Table 3-43. FSUB/FSUBP/FISUB Results . . . 3-399
Table 3-44. FSUBR/FSUBRP/FISUBR Results . . . 3-403
Table 3-45. FTST Results . . . 3-406
Table 3-46. FUCOM/FUCOMP/FUCOMPP Results . . . 3-408
Table 3-47. FXAM Results . . . 3-411
Table 3-48. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region . . . 3-418
Table 3-49. Field Definitions . . . 3-420
Table 3-50. Recreating FSAVE Format . . . 3-422
Table 3-51. Layout of the 64-bit-mode FXSAVE Map with Promoted OperandSize . . . 3-423
Table 3-52. Layout of the 64-bit-mode FXSAVE Map with Default OperandSize . . . 3-424
Table 3-53. FYL2X Results . . . 3-431
Table 3-54. FYL2XP1 Results . . . 3-433
Table 3-55. IDIV Results . . . 3-454
Table 3-56. Decision Table . . . 3-472
Table 3-57. Segment and Gate Types . . . 3-521
Table 3-58. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes . . . 3-535
Table 3-59. 64-bit Mode LEA Operation with Address and Operand Size Attributes . . . 3-536
Table 3-60. Segment and Gate Descriptor Types . . . 3-559
Table 3-61. MUL Results . . . 3-695
Table 3-62. MWAIT Extension Register (ECX) . . . 3-711
Table 3-63. MWAIT Hints Register (EAX) . . . 3-712
Table 4-1. Recommended Multi-Byte Sequence of NOP Instruction . . . 4-5
Table 4-2. Valid General and Special Purpose Performance Counter Index Range for RDPMC . . . 4-247
Table 4-3. Repeat Prefixes . . . 4-256
Table 4-4. Decision Table for STI Results . . . 4-339
Table 4-5. SWAPGS Operation Parameters . . . 4-365
Table 4-6. MSRs Used By the SYSENTER and SYSEXIT Instructions . . . 4-369
Table 6-1. Layout of IA32_FEATURE_CONTROL . . . 5-2
Table 6-2. GETSEC Leaf Functions . . . 5-3
Table 6-3. GETSEC Capability Result Encoding (EBX = 0) . . . 5-9
Table 6-4. Register State Initialization after GETSEC[ENTERACCS] . . . 5-15
Table 6-5. IA32_MISC_ENALBES MSR Initialization by ENTERACCS and SENTER . . . 5-17
Table 6-6. Register State Initialization after GETSEC[SENTER] and GETSEC[WAKEUP] . . . 5-31
Table 6-7. SMX Reporting Parameters Format . . . 5-43
Table 6-8. External Memory Types Using Parameter 3 . . . 5-45
Table 6-9. Default Parameter Values . . . 5-46
Table 6-10. Supported Actions for GETSEC[SMCTRL(0)] . . . 5-49
Table 6-11. RLP MVMM JOIN Data Structure . . . 5-52

PAGE

Table A-1. Superscripts Utilized in Opcode Tables ..... A-7
Table A-2. One-byte Opcode Map: (00H — F7H) *..... A-9
Table A-3. Two-byte Opcode Map: 00H — 77H (First Byte is 0FH) * ..... A-11
Table A-4. Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) * ..... A-15
Table A-5. Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) *..... A-17
Table A-6. Opcode Extensions for One- and Two-byte Opcodes by Group Number *..... A-20
Table A-7. D8 Opcode Map When ModR/M Byte is Within 00H to BFH *..... A-23
Table A-8. D8 Opcode Map When ModR/M Byte is Outside 00H to BFH *..... A-23
Table A-9. D9 Opcode Map When ModR/M Byte is Within 00H to BFH *..... A-24
Table A-10. D9 Opcode Map When ModR/M Byte is Outside 00H to BFH *..... A-24
Table A-11. DA Opcode Map When ModR/M Byte is Within 00H to BFH *..... A-25
Table A-12. DA Opcode Map When ModR/M Byte is Outside 00H to BFH *..... A-25
Table A-13. DB Opcode Map When ModR/M Byte is Within 00H to BFH *..... A-26
Table A-14. DB Opcode Map When ModR/M Byte is Outside 00H to BFH *..... A-26
Table A-15. DC Opcode Map When ModR/M Byte is Within 00H to BFH *..... A-27
Table A-16. DC Opcode Map When ModR/M Byte is Outside 00H to BFH *..... A-27
Table A-17. DD Opcode Map When ModR/M Byte is Within 00H to BFH *..... A-28
Table A-18. DD Opcode Map When ModR/M Byte is Outside 00H to BFH *..... A-28
Table A-19. DE Opcode Map When ModR/M Byte is Within 00H to BFH *..... A-29
Table A-20. DE Opcode Map When ModR/M Byte is Outside 00H to BFH *..... A-29
Table A-21. DF Opcode Map When ModR/M Byte is Within 00H to BFH *..... A-30
Table A-22. DF Opcode Map When ModR/M Byte is Outside 00H to BFH *..... A-30
Table B-1. Special Fields Within Instruction Encodings ..... B-3
Table B-2. Encoding of reg Field When w Field is Not Present in Instruction..... B-3
Table B-4. Encoding of reg Field When w Field is Not Present in Instruction..... B-4
Table B-3. Encoding of reg Field When w Field is Present in Instruction..... B-4
Table B-5. Encoding of reg Field When w Field is Present in Instruction..... B-5
Table B-6. Encoding of Operand Size (w) Bit ..... B-5
Table B-7. Encoding of Sign-Extend (s) Bit ..... B-6
Table B-8. Encoding of the Segment Register (sreg) Field..... B-6
Table B-9. Encoding of Special-Purpose Register (eee) Field ..... B-7
Table B-11. Encoding of Operation Direction (d) Bit..... B-8
Table B-10. Encoding of Conditional Test (tttn) Field..... B-8
Table B-12. Notes on Instruction Encoding..... B-9
Table B-13. General Purpose Instruction Formats and Encodings for Non-64-Bit Modes..... B-9
Table B-14. Special Symbols..... B-24
Table B-15. General Purpose Instruction Formats and Encodings for 64-Bit Mode..... B-24
Table B-16. Pentium Processor Family Instruction Formats and Encodings, Non-64-Bit Modes ..... B-53
Table B-17. Pentium Processor Family Instruction Formats and Encodings, 64-Bit Mode .... B-53
Table B-18. Encoding of Granularity of Data Field (gg) ..... B-54
Table B-19. MMX Instruction Formats and Encodings ..... B-55
Table B-20. Formats and Encodings of P6 Family Instructions..... B-58
Table B-21. Formats and Encodings of SSE Floating-Point Instructions..... B-60
Table B-22. Formats and Encodings of SSE Integer Instructions ..... B-66

PAGE

Table B-23. Format and Encoding of SSE Cacheability & Memory Ordering Instructions.....B-67
Table B-24. Encoding of Granularity of Data Field (gg).....B-68
Table B-25. Formats and Encodings of SSE2 Floating-Point Instructions.....B-69
Table B-26. Formats and Encodings of SSE2 Integer Instructions .....B-77
Table B-27. Format and Encoding of SSE2 Cacheability Instructions.....B-84
Table B-28. Formats and Encodings of SSE3 Floating-Point Instructions.....B-85
Table B-29. Formats and Encodings for SSE3 Event Management Instructions.....B-86
Table B-30. Formats and Encodings for SSE3 Integer and Move Instructions.....B-86
Table B-31. Formats and Encodings for SSSE3 Instructions.....B-87
Table B-32. Special Case Instructions Promoted Using REX.W.....B-91
Table B-33. General Floating-Point Instruction Formats .....B-95
Table B-34. Floating-Point Instruction Formats and Encodings.....B-96
Table B-35. Encodings for VMX Instructions ..... B-101
Table B-36. Encodings for SMX Instructions ..... B-103
Table C-1. Simple Intrinsics..... C-3
Table C-2. Composite Intrinsics .....C-14

CONTENTS

PAGE

# CHAPTER 1
# ABOUT THIS MANUAL

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A & 2B*: *Instruction Set Reference* (order numbers 253666 and 253667) are part of a set that describes the architecture and programming environment of all Intel 64 and IA-32 architecture processors. Other volumes in this set are:

- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (Order Number 253665).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A & 3B: System Programming Guide* (order numbers 253668 and 253669).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1,* describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A & 2B*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A & 3B,* describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, addresses the programming environment for classes of software that host operating systems.

## 1.1 IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV

- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor
- Intel® Core™2 Extreme Quad-core processor
- Intel® Xeon® processor 7100, 7300 series
- Intel® Pentium® Dual-Core processor

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5300, 7100, 7300 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors.

Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

## 1.2 OVERVIEW OF VOLUME 2A AND 2B: INSTRUCTION SET REFERENCE

A description of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A & 2B*, content follows:

**Chapter 1 — About This Manual.** Gives an overview of all five volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel® manuals and documentation of interest to programmers and hardware designers.

**Chapter 2 — Instruction Format.** Describes the machine-level instruction format used for all IA-32 instructions and gives the allowable encodings of prefixes, the operand-identifier byte (ModR/M byte), the addressing-mode specifier byte (SIB byte), and the displacement and immediate bytes.

**Chapter 3 — Instruction Set Reference, A-M.** Describes Intel 64 and IA-32 instructions in detail, including an algorithmic description of operations, the effect on flags, the effect of operand- and address-size attributes, and the exceptions that may be generated. The instructions are arranged in alphabetical order. General-purpose, x87 FPU, Intel MMX™ technology, SSE/SSE2/SSE3 extensions, and system instructions are included.

**Chapter 4 — Instruction Set Reference, N-Z.** Continues the description of Intel 64 and IA-32 instructions started in Chapter 3. It provides the balance of the alphabetized list of instructions and starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

**Chapter 5 — VMX Instruction Reference.** Describes the virtual-machine extensions (VMX). VMX is intended for a system executive to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments.

**Chapter 6— Safer Mode Extensions Reference.** Describes the safer mode extensions (SMX). SMX is intended for a system executive to support launching a measured environment in a platform where the identity of the software controlling the platform hardware can be measured for the purpose of making trust decisions.

**Appendix A — Opcode Map.** Gives an opcode map for the IA-32 instruction set.

**Appendix B — Instruction Formats and Encodings.** Gives the binary encoding of each form of each IA-32 instruction.

**Appendix C — Intel® C/C++Compiler Intrinsics and Functional Equivalents.** Lists the Intel® C/C++compiler intrinsics and their assembly code equivalents for each of the IA-32 MMX and SSE/SSE2/SSE3 instructions.

## 1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

### 1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

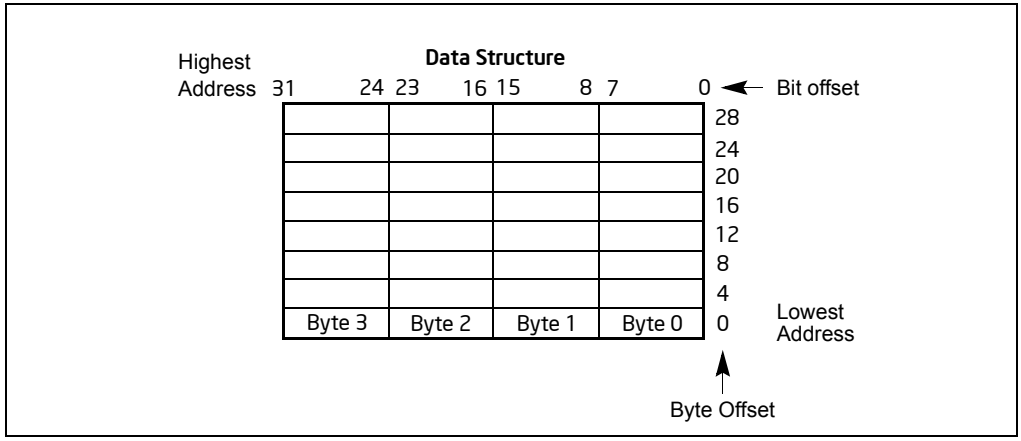



Figure 1-1. Bit and Byte Order

## 1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

### NOTE

Avoid any software dependence upon the state of reserved bits in IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

### 1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands *argument1*, *argument2*, and *argument3* are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

### 1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The “B” designation is only used in situations where confusion as to the type of number might arise.

### 1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes in memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**.

For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

### 1.3.6 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

### 1.3.7 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.

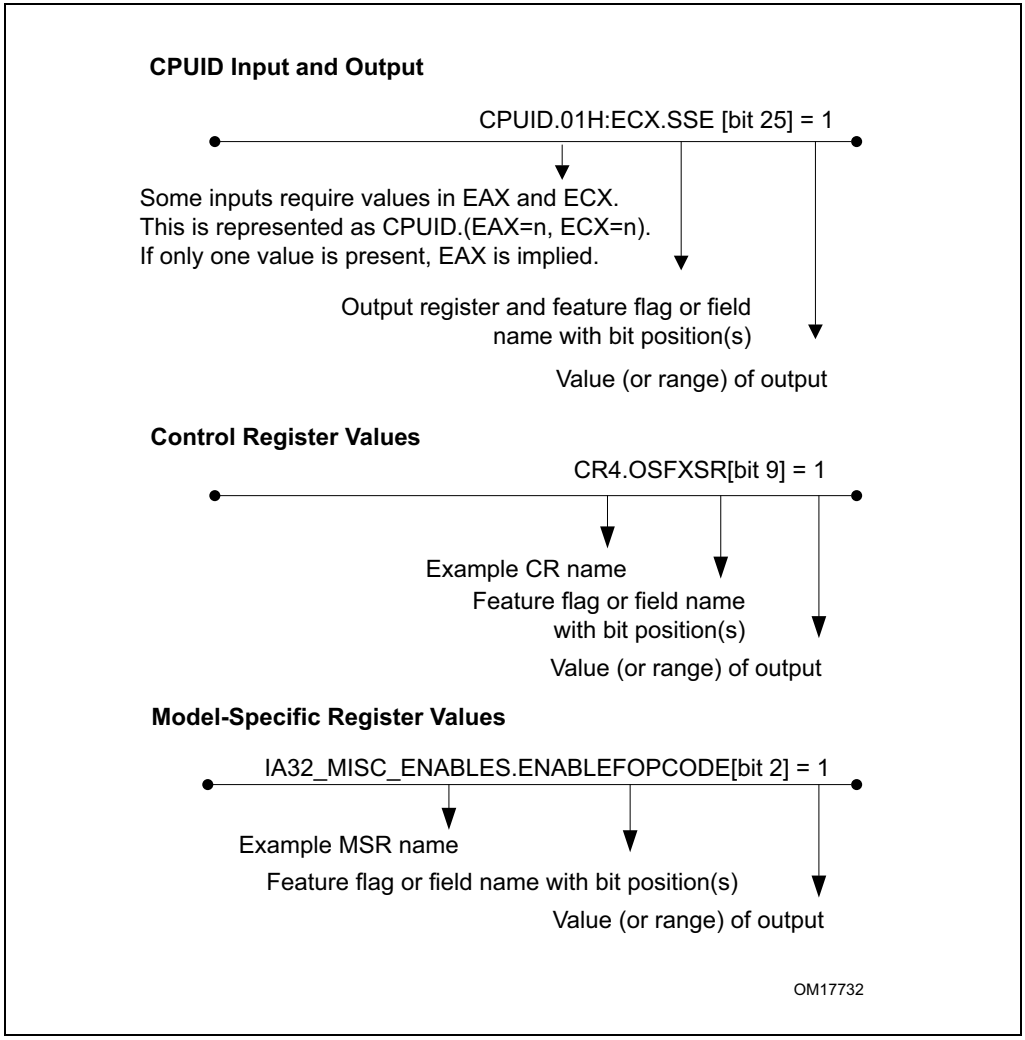



Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

## 1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed on-line at:

http://developer.intel.com/products/processor/manuals/index.htm

Some of the documents listed at this web site can be viewed on-line; others can be ordered. The literature available is listed by Intel processor and then by the following

literature types: applications notes, data sheets, manuals, papers, and specification updates.

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help
  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® Fortran Compiler documentation and online help
  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® VTune™ Performance Analyzer documentation and online help
  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in five volumes)
  http://developer.intel.com/products/processor/manuals/index.htm
- Intel® 64 and IA-32 Architectures Optimization Reference Manual
  http://developer.intel.com/products/processor/manuals/index.htm
- Intel® Processor Identification with the CPUID Instruction, AP-485
  http://www.intel.com/support/processors/sb/cs-009861.htm
- TLBs, Paging-Structure Caches, and Their Invalidation,
  http://developer.intel.com/products/processor/manuals/index.htm
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide, http://www.intel.com/technology/security/index.htm
- Intel® SSE4 Programming Reference,
  http://developer.intel.com/products/processor/manuals/index.htm
- Developing Multi-threaded Applications: A Platform Consistent Approach
  http://cache-www.intel.com/cd/00/00/05/15/51534_developing_multithreaded_applications.pdf
- Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor MP
  http://www3.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/knowledgebase/19083.htm

More relevant links are:

- Software network link:

  http://softwarecommunity.intel.com/isn/home/
- Developer centers:

  http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/index.htm
- Processor support general link:

  http://www.intel.com/support/processors/
- Software products and packages:

  http://www.intel.com/cd/software/products/asmo-na/eng/index.htm

- Intel 64 and IA-32 processor manuals (printed or PDF downloads):
  http://developer.intel.com/products/processor/manuals/index.htm
- Intel® Multi-Core Technology:
  http://developer.intel.com/multi-core/index.htm
- Hyper-Threading Technology (HT Technology):
  http://developer.intel.com/technology/hyperthread/

# CHAPTER 2
# INSTRUCTION FORMAT

This chapter describes the instruction format for all Intel 64 and IA-32 processors. The instruction format for protected mode, real-address mode and virtual-8086 mode is described in Section 2.1. Increments provided for IA-32e mode and its sub-modes are described in Section 2.2.

## 2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

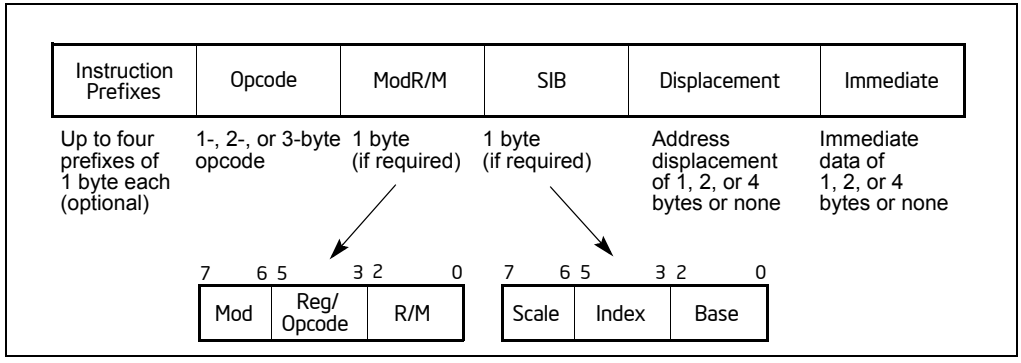



Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

### 2.1.1 Instruction Prefixes

Instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, one prefix may be used from each of four groups (Groups 1, 2, 3, 4) and be placed in any order.

- Group 1
  - — Lock and repeat prefixes:
    - F0H—LOCK

    - F2H—REPNE/REPNZ (used only with string instructions; when used with the escape opcode 0FH, this prefix is treated as a mandatory prefix for some SIMD instructions)
    - F3H—REP or REPE/REPZ (used only with string instructions; when used with the escape opcode 0FH, this prefix is treated as an mandatory prefix for some SIMD instructions)
- Group 2
  - Segment override prefixes:
    - 2EH—CS segment override (use with any branch instruction is reserved)
    - 36H—SS segment override prefix (use with any branch instruction is reserved)
    - 3EH—DS segment override prefix (use with any branch instruction is reserved)
    - 26H—ES segment override prefix (use with any branch instruction is reserved)
    - 64H—FS segment override prefix (use with any branch instruction is reserved)
    - 65H—GS segment override prefix (use with any branch instruction is reserved)
  - Branch hints:
    - 2EH—Branch not taken (used only with *Jcc* instructions)
    - 3EH—Branch taken (used only with *Jcc* instructions)
- Group 3
    - 66H—Operand-size override prefix (when used with the escape opcode 0FH, this is treated as a mandatory prefix for some SIMD instructions)
- Group 4
    - 67H—Address-size override prefix

The LOCK prefix (F0H) forces an operation that ensures exclusive use of shared memory in a multiprocessor environment. See “LOCK—Assert LOCK# Signal Prefix” in Chapter 3, “Instruction Set Reference, A-M,” for a description of this prefix.

Repeat prefixes (F2H, F3H) cause an instruction to be repeated for each element of a string. Use these prefixes only with string instructions (MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS). Their use, followed by 0FH, is treated as a mandatory prefix by a number of SSE/SSE2/SSE3 instructions. Use of repeat prefixes and/or undefined opcodes with other Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch. Use these prefixes only with conditional branch instructions (*Jcc*). Other use of branch hint prefixes and/or other undefined

opcodes with Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size. Use of 66H followed by 0FH is treated as a mandatory prefix by some SSE/SSE2/SSE3 instructions. Other use of the 66H prefix with MMX/SSE/SSE2/SSE3 instructions is reserved; such use may cause unpredictable behavior.

The address-size override prefix (67H) allows programs to switch between 16- and 32-bit addressing. Either size can be the default; the prefix selects the non-default size. Using this prefix and/or other undefined opcodes when operands for the instruction do not reside in memory is reserved; such use may cause unpredictable behavior.

### 2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation, size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte, or
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet)

For example, CVTDQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix for SSE/SSE2/SSE3 instructions (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes, or
- A mandatory prefix (66H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet)

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

### 2.1.3 ModR/M and SIB Bytes

Many instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary opcode. The ModR/M byte contains three fields of information:

- The *mod* field combines with the r/m field to form 32 possible values: eight registers and 24 addressing modes.
- The *reg/opcode* field specifies either a register number or three more bits of opcode information. The purpose of the reg/opcode field is specified in the primary opcode.
- The *r/m* field can specify a register as an operand or it can be combined with the mod field to encode an addressing mode. Sometimes, certain combinations of the *mod* field and the *r/m* field is used to express opcode information for some instructions.

Certain encodings of the ModR/M byte require a second addressing byte (the SIB byte). The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:

- The *scale* field specifies the scale factor.
- The *index* field specifies the register number of the index register.
- The *base* field specifies the register number of the base register.

See Section 2.1.5 for the encodings of the ModR/M and SIB bytes.

### 2.1.4 Displacement and Immediate Bytes

Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required; it be 1, 2, or 4 bytes.

If an instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.

### 2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes

The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Table 2-1 through Table 2-3: 16-bit addressing forms specified by the ModR/M byte are in Table 2-1 and 32-bit addressing forms are in Table 2-2. Table 2-3 shows 32-bit addressing forms specified by the SIB byte. In cases where the reg/opcode field in the ModR/M byte represents an extended opcode, valid encodings are shown in Appendix B.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 options provide ways of specifying a memory

location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology and XMM registers.

The Mod and R/M columns in Table 2-1 and Table 2-2 give the binary encodings of the Mod and R/M fields required to obtain the effective address listed in the first column. For example: see the row indicated by Mod = 11B, R/M = 000B. The row identifies the general-purpose registers EAX, AX or AL; MMX technology register MM0; or XMM register XMM0. The register used is determined by the opcode byte and the operand-size attribute.

Now look at the seventh row in either table (labeled "REG ="). This row specifies the use of the 3-bit Reg/Opcode field when the field is used to give the location of a second operand. The second operand must be a general-purpose, MMX technology, or XMM register. Rows one through five list the registers that may correspond to the value in the table. Again, the register used is determined by the opcode byte along with the operand-size attribute.

If the instruction does not require a second operand, then the Reg/Opcode field may be used as an opcode extension. This use is represented by the sixth row in the tables (labeled "/digit (Opcode)"). Note that values in row six are represented in decimal form.

The body of Table 2-1 and Table 2-2 (under the label "Value of ModR/M Byte (in Hexadecimal)") contains a 32 by 8 array that presents all of 256 values of the ModR/M byte (in hexadecimal). Bits 3, 4 and 5 are specified by the column of the table in which a byte resides. The row specifies bits 0, 1 and 2; and bits 6 and 7. The figure below demonstrates interpretation of one table value.

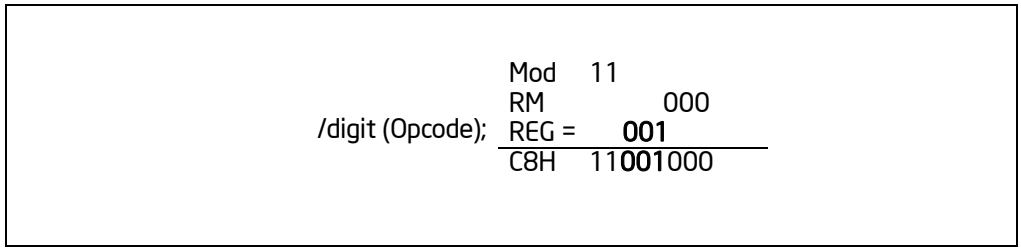



**Figure 2-2. Table Interpretation of ModR/M Byte (C8H)**

## Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

| r8(/r)<br>r16(/r)<br>r32(/r)<br>mm(/r)<br>xmm(/r)<br>(In decimal) /digit (Opcode)<br>(In binary) REG = | | | AL<br>AX<br>EAX<br>MM0<br>XMM0<br>0<br>000 | CL<br>CX<br>ECX<br>MM1<br>XMM1<br>1<br>001 | DL<br>DX<br>EDX<br>MM2<br>XMM2<br>2<br>010 | BL<br>BX<br>EBX<br>MM3<br>XMM3<br>3<br>011 | AH<br>SP<br>ESP<br>MM4<br>XMM4<br>4<br>100 | CH<br>BP1<br>EBP<br>MM5<br>XMM5<br>5<br>101 | DH<br>SI<br>ESI<br>MM6<br>XMM6<br>6<br>110 | BH<br>DI<br>EDI<br>MM7<br>XMM7<br>7<br>111 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Effective Address** | **Mod** | **R/M** | **Value of ModR/M Byte (in Hexadecimal)** | | | | | | | |
| [BX+SI]<br>[BX+DI]<br>[BP+SI]<br>[BP+DI]<br>[SI]<br>[DI]<br>disp16[2]<br>[BX] | 00 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 00<br>01<br>02<br>03<br>04<br>05<br>06<br>07 | 08<br>09<br>0A<br>0B<br>0C<br>0D<br>0E<br>0F | 10<br>11<br>12<br>13<br>14<br>15<br>16<br>17 | 18<br>19<br>1A<br>1B<br>1C<br>1D<br>1E<br>1F | 20<br>21<br>22<br>23<br>24<br>25<br>26<br>27 | 28<br>29<br>2A<br>2B<br>2C<br>2D<br>2E<br>2F | 30<br>31<br>32<br>33<br>34<br>35<br>36<br>37 | 38<br>39<br>3A<br>3B<br>3C<br>3D<br>3E<br>3F |
| [BX+SI]+disp8[3]<br>[BX+DI]+disp8<br>[BP+SI]+disp8<br>[BP+DI]+disp8<br>[SI]+disp8<br>[DI]+disp8<br>[BP]+disp8<br>[BX]+disp8 | 01 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 40<br>41<br>42<br>43<br>44<br>45<br>46<br>47 | 48<br>49<br>4A<br>4B<br>4C<br>4D<br>4E<br>4F | 50<br>51<br>52<br>53<br>54<br>55<br>56<br>57 | 58<br>59<br>5A<br>5B<br>5C<br>5D<br>5E<br>5F | 60<br>61<br>62<br>63<br>64<br>65<br>66<br>67 | 68<br>69<br>6A<br>6B<br>6C<br>6D<br>6E<br>6F | 70<br>71<br>72<br>73<br>74<br>75<br>76<br>77 | 78<br>79<br>7A<br>7B<br>7C<br>7D<br>7E<br>7F |
| [BX+SI]+disp16<br>[BX+DI]+disp16<br>[BP+SI]+disp16<br>[BP+DI]+disp16<br>[SI]+disp16<br>[DI]+disp16<br>[BP]+disp16<br>[BX]+disp16 | 10 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 80<br>81<br>82<br>83<br>84<br>85<br>86<br>87 | 88<br>89<br>8A<br>8B<br>8C<br>8D<br>8E<br>8F | 90<br>91<br>92<br>93<br>94<br>95<br>96<br>97 | 98<br>99<br>9A<br>9B<br>9C<br>9D<br>9E<br>9F | A0<br>A1<br>A2<br>A3<br>A4<br>A5<br>A6<br>A7 | A8<br>A9<br>AA<br>AB<br>AC<br>AD<br>AE<br>AF | B0<br>B1<br>B2<br>B3<br>B4<br>B5<br>B6<br>B7 | B8<br>B9<br>BA<br>BB<br>BC<br>BD<br>BE<br>BF |
| EAX/AX/AL/MM0/XMM0<br>ECX/CX/CL/MM1/XMM1<br>EDX/DX/DL/MM2/XMM2<br>EBX/BX/BL/MM3/XMM3<br>ESP/SP/AHMM4/XMM4<br>EBP/BP/CH/MM5/XMM5<br>ESI/SI/DH/MM6/XMM6<br>EDI/DI/BH/MM7/XMM7 | 11 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | C0<br>C1<br>C2<br>C3<br>C4<br>C5<br>C6<br>C7 | C8<br>C9<br>CA<br>CB<br>CC<br>CD<br>CE<br>CF | D0<br>D1<br>D2<br>D3<br>D4<br>D5<br>D6<br>D7 | D8<br>D9<br>DA<br>DB<br>DC<br>DD<br>DE<br>DF | E0<br>EQ<br>E2<br>E3<br>E4<br>E5<br>E6<br>E7 | E8<br>E9<br>EA<br>EB<br>EC<br>ED<br>EE<br>EF | F0<br>F1<br>F2<br>F3<br>F4<br>F5<br>F6<br>F7 | F8<br>F9<br>FA<br>FB<br>FC<br>FD<br>FE<br>FF |

**NOTES:**

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.

## Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

| r8(/r)<br>r16(/r)<br>r32(/r)<br>mm(/r)<br>xmm(/r)<br>(In decimal) /digit (Opcode)<br>(In binary) REG = | | | AL<br>AX<br>EAX<br>MM0<br>XMM0<br>0<br>000 | CL<br>CX<br>ECX<br>MM1<br>XMM1<br>1<br>001 | DL<br>DX<br>EDX<br>MM2<br>XMM2<br>2<br>010 | BL<br>BX<br>EBX<br>MM3<br>XMM3<br>3<br>011 | AH<br>SP<br>ESP<br>MM4<br>XMM4<br>4<br>100 | CH<br>BP<br>EBP<br>MM5<br>XMM5<br>5<br>101 | DH<br>SI<br>ESI<br>MM6<br>XMM6<br>6<br>110 | BH<br>DI<br>EDI<br>MM7<br>XMM7<br>7<br>111 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Effective Address** | **Mod** | **R/M** | **Value of ModR/M Byte (in Hexadecimal)** | | | | | | | |
| [EAX]<br>[ECX]<br>[EDX]<br>[EBX]<br>[--][--][1]<br>disp32[2]<br>[ESI]<br>[EDI] | 00 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 00<br>01<br>02<br>03<br>04<br>05<br>06<br>07 | 08<br>09<br>0A<br>0B<br>0C<br>0D<br>0E<br>0F | 10<br>11<br>12<br>13<br>14<br>15<br>16<br>17 | 18<br>19<br>1A<br>1B<br>1C<br>1D<br>1E<br>1F | 20<br>21<br>22<br>23<br>24<br>25<br>26<br>27 | 28<br>29<br>2A<br>2B<br>2C<br>2D<br>2E<br>2F | 30<br>31<br>32<br>33<br>34<br>35<br>36<br>37 | 38<br>39<br>3A<br>3B<br>3C<br>3D<br>3E<br>3F |
| [EAX]+disp8[3]<br>[ECX]+disp8<br>[EDX]+disp8<br>[EBX]+disp8<br>[--][--]+disp8<br>[EBP]+disp8<br>[ESI]+disp8<br>[EDI]+disp8 | 01 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 40<br>41<br>42<br>43<br>44<br>45<br>46<br>47 | 48<br>49<br>4A<br>4B<br>4C<br>4D<br>4E<br>4F | 50<br>51<br>52<br>53<br>54<br>55<br>56<br>57 | 58<br>59<br>5A<br>5B<br>5C<br>5D<br>5E<br>5F | 60<br>61<br>62<br>63<br>64<br>65<br>66<br>67 | 68<br>69<br>6A<br>6B<br>6C<br>6D<br>6E<br>6F | 70<br>71<br>72<br>73<br>74<br>75<br>76<br>77 | 78<br>79<br>7A<br>7B<br>7C<br>7D<br>7E<br>7F |
| [EAX]+disp32<br>[ECX]+disp32<br>[EDX]+disp32<br>[EBX]+disp32<br>[--][--]+disp32<br>[EBP]+disp32<br>[ESI]+disp32<br>[EDI]+disp32 | 10 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | 80<br>81<br>82<br>83<br>84<br>85<br>86<br>87 | 88<br>89<br>8A<br>8B<br>8C<br>8D<br>8E<br>8F | 90<br>91<br>92<br>93<br>94<br>95<br>96<br>97 | 98<br>99<br>9A<br>9B<br>9C<br>9D<br>9E<br>9F | A0<br>A1<br>A2<br>A3<br>A4<br>A5<br>A6<br>A7 | A8<br>A9<br>AA<br>AB<br>AC<br>AD<br>AE<br>AF | B0<br>B1<br>B2<br>B3<br>B4<br>B5<br>B6<br>B7 | B8<br>B9<br>BA<br>BB<br>BC<br>BD<br>BE<br>BF |
| EAX/AX/AL/MM0/XMM0<br>ECX/CX/CL/MM/XMM1<br>EDX/DX/DL/MM2/XMM2<br>EBX/BX/BL/MM3/XMM3<br>ESP/SP/AH/MM4/XMM4<br>EBP/BP/CH/MM5/XMM5<br>ESI/SI/DH/MM6/XMM6<br>EDI/DI/BH/MM7/XMM7 | 11 | 000<br>001<br>010<br>011<br>100<br>101<br>110<br>111 | C0<br>C1<br>C2<br>C3<br>C4<br>C5<br>C6<br>C7 | C8<br>C9<br>CA<br>CB<br>CC<br>CD<br>CE<br>CF | D0<br>D1<br>D2<br>D3<br>D4<br>D5<br>D6<br>D7 | D8<br>D9<br>DA<br>DB<br>DC<br>DD<br>DE<br>DF | E0<br>E1<br>E2<br>E3<br>E4<br>E5<br>E6<br>E7 | E8<br>E9<br>EA<br>EB<br>EC<br>ED<br>EE<br>EF | F0<br>F1<br>F2<br>F3<br>F4<br>F5<br>F6<br>F7 | F8<br>F9<br>FA<br>FB<br>FC<br>FD<br>FE<br>FF |

**NOTES:**

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. Table rows in the body

of the table indicate the register used as the index (SIB byte bits 3, 4 and 5) and the scaling factor (determined by SIB byte bits 6 and 7).

**Table 2-3. 32-Bit Addressing Forms with the SIB Byte**

| r32<br>(In decimal) Base =<br>(In binary) Base = | | | EAX<br>0<br>000 | ECX<br>1<br>001 | EDX<br>2<br>010 | EBX<br>3<br>011 | ESP<br>4<br>100 | [*]<br>5<br>101 | ESI<br>6<br>110 | EDI<br>7<br>111 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Scaled Index** | **SS** | **Index** | **Value of SIB Byte (in Hexadecimal)** | | | | | | | |
| [EAX] | 00 | 000 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| [ECX] | | 001 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
| [EDX] | | 010 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| [EBX] | | 011 | 18 | 19 | 1A | 1B | 1C | 1D | 1E | 1F |
| none | | 100 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| [EBP] | | 101 | 28 | 29 | 2A | 2B | 2C | 2D | 2E | 2F |
| [ESI] | | 110 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| [EDI] | | 111 | 38 | 39 | 3A | 3B | 3C | 3D | 3E | 3F |
| [EAX*2] | 01 | 000 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| [ECX*2] | | 001 | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F |
| [EDX*2] | | 010 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| [EBX*2] | | 011 | 58 | 59 | 5A | 5B | 5C | 5D | 5E | 5F |
| none | | 100 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| [EBP*2] | | 101 | 68 | 69 | 6A | 6B | 6C | 6D | 6E | 6F |
| [ESI*2] | | 110 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |
| [EDI*2] | | 111 | 78 | 79 | 7A | 7B | 7C | 7D | 7E | 7F |
| [EAX*4] | 10 | 000 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 |
| [ECX*4] | | 001 | 88 | 89 | 8A | 8B | 8C | 8D | 8E | 8F |
| [EDX*4] | | 010 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 |
| [EBX*4] | | 011 | 98 | 89 | 9A | 9B | 9C | 9D | 9E | 9F |
| none | | 100 | A0 | A1 | A2 | A3 | A4 | A5 | A6 | A7 |
| [EBP*4] | | 101 | A8 | A9 | AA | AB | AC | AD | AE | AF |
| [ESI*4] | | 110 | B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 |
| [EDI*4] | | 111 | B8 | B9 | BA | BB | BC | BD | BE | BF |
| [EAX*8] | 11 | 000 | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
| [ECX*8] | | 001 | C8 | C9 | CA | CB | CC | CD | CE | CF |
| [EDX*8] | | 010 | D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7 |
| [EBX*8] | | 011 | D8 | D9 | DA | DB | DC | DD | DE | DF |
| none | | 100 | E0 | E1 | E2 | E3 | E4 | E5 | E6 | E7 |
| [EBP*8] | | 101 | E8 | E9 | EA | EB | EC | ED | EE | EF |
| [ESI*8] | | 110 | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 |
| [EDI*8] | | 111 | F8 | F9 | FA | FB | FC | FD | FE | FF |

**NOTES:**

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

| MOD bits | Effective Address |
|---|---|
| 00 | [scaled index] + disp32 |
| 01 | [scaled index] + disp8 + [EBP] |
| 10 | [scaled index] + disp32 + [EBP] |

## 2.2 IA-32E MODE

IA-32e mode has two sub-modes. These are:

- **Compatibility Mode.** Enables a 64-bit operating system to run most legacy protected mode software unmodified.
- **64-Bit Mode.** Enables a 64-bit operating system to run applications written to access 64-bit address space.

### 2.2.1 REX Prefixes

REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:

- Specify GPRs and SSE registers.
- Specify 64-bit operand size.
- Specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored.

Only one REX prefix is allowed per instruction. If used, the prefix must immediately precede the opcode byte or the two-byte opcode escape prefix (if present). Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.

| Legacy Prefixes | REX Prefix | Opcode | ModR/M | SIB | Displacement | Immediate |
|---|---|---|---|---|---|---|
| Grp 1, Grp 2, Grp 3, Grp 4 (optional) | (optional) | 1-, 2-, or 3-byte opcode | 1 byte (if required) | 1 byte (if required) | Address displacement of 1, 2, or 4 bytes | Immediate data of 1, 2, or 4 bytes or none |

**Figure 2-3. Prefix Ordering in 64-bit Mode**

#### 2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:

- ModR/M: the reg and r/m fields of the ModR/M byte
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte
- Instructions without ModR/M: the reg field of the opcode

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

#### 2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode form of INC/DEC instruction not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 though Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:

- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.
- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.
- REX.B either modifies the base in the ModR/M r/m field or SIB base field; or it modifies the opcode reg field used for accessing GPRs.

Table 2-4. REX Prefix Fields [BITS: 0100WRXB]

| Field Name | Bit Position | Definition |
|---|---|---|
| - | 7:4 | 0100 |
| W | 3 | 0 = Operand size determined by CS.D |
| | | 1 = 64 Bit Operand Size |
| R | 2 | Extension of the ModR/M reg field |
| X | 1 | Extension of the SIB index field |
| B | 0 | Extension of the ModR/M r/m field, SIB base field, or Opcode reg field |

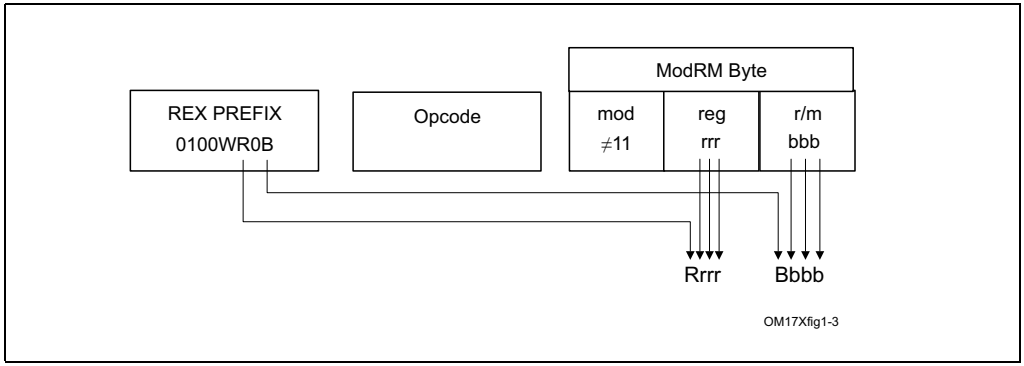



Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used

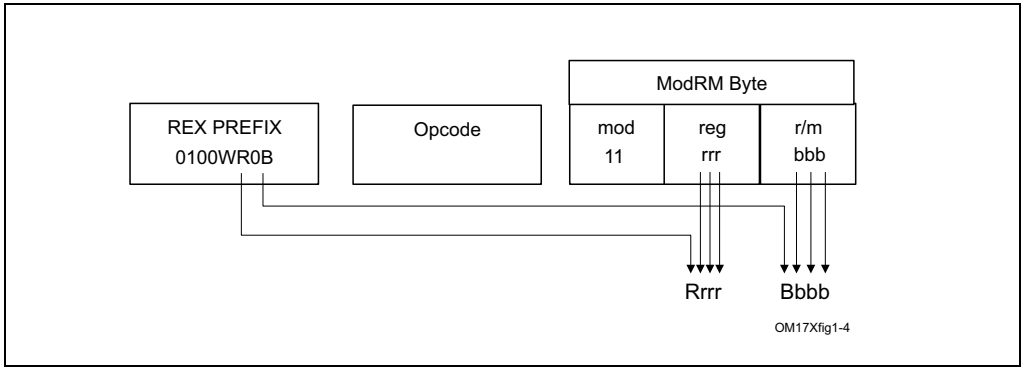



Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used

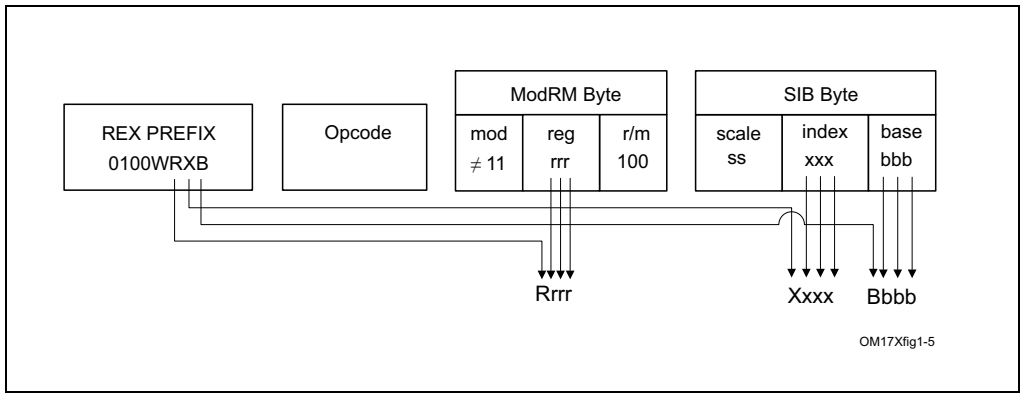



Figure 2-6. Memory Addressing With a SIB Byte

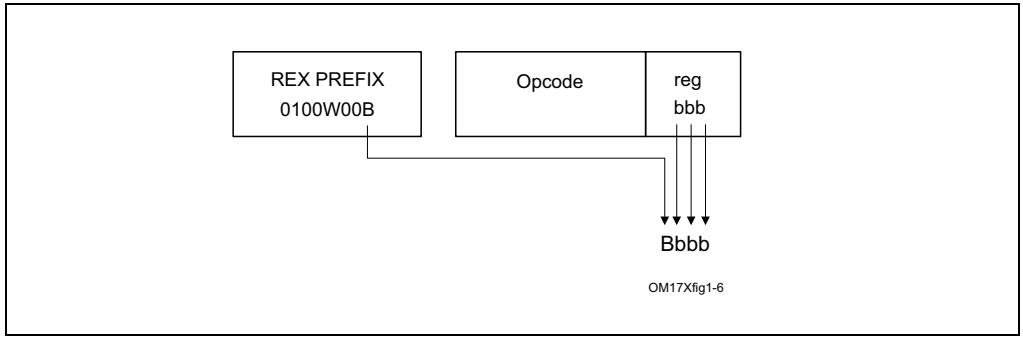



Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

In the IA-32 architecture, byte registers (AH, AL, BH, BL, CH, CL, DH, and DL) are encoded in the ModR/M byte's reg field, the r/m field or the opcode reg field as registers 0 through 7. REX prefixes provide an additional addressing capability for byte-registers that makes the least-significant byte of GPRs available for byte operations.

Certain combinations of the fields of the ModR/M byte and the SIB byte have special meaning for register encodings. For some combinations, fields expanded by the REX prefix are not decoded. Table 2-5 describes how each case behaves.

Table 2-5. Special Cases of REX Encodings

| ModR/M or SIB | Sub-field Encodings | Compatibility Mode Operation | Compatibility Mode Implications | Additional Implications |
|---|---|---|---|---|
| ModR/M Byte | mod != 11<br>r/m == b*100(ESP) | SIB byte present. | SIB byte required for ESP-based addressing. | REX prefix adds a fourth bit (b) which is not decoded (don't care).<br>SIB byte also required for R12-based addressing. |
| ModR/M Byte | mod == 0<br>r/m == b*101(EBP) | Base register not used. | EBP without a displacement must be done using<br>mod = 01 with displacement of 0. | REX prefix adds a fourth bit (b) which is not decoded (don't care).<br>Using RBP or R13 without displacement must be done using mod = 01 with a displacement of 0. |
| SIB Byte | index == 0100(ESP) | Index register not used. | ESP cannot be used as an index register. | REX prefix adds a fourth bit (b) which is decoded.<br>There are no additional implications. The expanded index field allows distinguishing RSP from R12, therefore R12 can be used as an index. |
| SIB Byte | base == 0101(EBP) | Base register is unused if mod = 0. | Base register depends on mod encoding. | REX prefix adds a fourth bit (b) which is not decoded.<br>This requires explicit displacement to be used with EBP/RBP or R13. |

NOTES:

* Don't care about value of REX.B

### 2.2.1.3 Displacement

Addressing in 64-bit mode uses existing 32-bit ModR/M and SIB encodings. The ModR/M and SIB displacement sizes do not change. They remain 8 bits or 32 bits and are sign-extended to 64 bits.

### 2.2.1.4 Direct Memory-Offset MOVs

In 64-bit mode, direct memory-offset forms of the MOV instruction are extended to specify a 64-bit immediate absolute address. This address is called a moffset. No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the

size of the memory offset follows the address-size default (64 bits in 64-bit mode). See Table 2-6.

**Table 2-6. Direct Memory Offset Form of MOV**

| Opcode | Instruction |
|---|---|
| A0 | MOV AL, moffset |
| A1 | MOV EAX, moffset |
| A2 | MOV moffset, AL |
| A3 | MOV moffset, EAX |

### 2.2.1.5 Immediates

In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use.

Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions (opcodes B8H – BFH) move 16-bits or 32-bits of immediate data (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits, these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default operand size to a 64-bit operand size.

For example:

```
48 B8  8877665544332211  MOV RAX,1122334455667788H
```

### 2.2.1.6 RIP-Relative Addressing

A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

In IA-32 architecture and compatibility mode, addressing relative to the instruction pointer is available only with control-transfer instructions. In 64-bit mode, instructions that use ModR/M addressing can use RIP-relative addressing. Without RIP-relative addressing, all ModR/M instruction modes address memory relative to zero.

RIP-relative addressing allows specific ModR/M modes to address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2GB from the RIP. Table 2-7 shows the ModR/M and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-addressing exist in the current ModR/M and SIB encodings. There is one ModR/M encoding and there are several SIB encodings. RIP-relative addressing is encoded using a redundant form.

In 64-bit mode, the ModR/M Disp32 (32-bit displacement) encoding is re-defined to be RIP+Disp32 rather than displacement-only. See Table 2-7.

Table 2-7. RIP-Relative Addressing

| ModR/M and SIB Sub-field Encodings | | Compatibility Mode Operation | 64-bit Mode Operation | Additional Implications in 64-bit mode |
|---|---|---|---|---|
| ModR/M Byte | mod == 00 | Disp32 | RIP + Disp32 | Must use SIB form with normal (zero-based) displacement addressing |
| | r/m == 101 (none) | | | |
| SIB Byte | base == 101 (none) | if mod = 00, Disp32 | Same as legacy | None |
| | index == 100 (none) | | | |
| | scale = 0, 1, 2, 4 | | | |

The ModR/M encoding for RIP-relative addressing does not depend on using prefix. Specifically, the r/m bit field encoding of 101B (used to select RIP-relative addressing) is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101B) with mod = 00B still results in RIP-relative addressing. The 4-bit r/m field of REX.B combined with ModR/M is not fully decoded. In order to address R13 with no displacement, software must encode R13 + 0 using a 1-byte displacement of zero.

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. The use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits.

#### 2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches
- All instructions, except far branches, that implicitly reference the RSP

### 2.2.2 Additional Encodings for Control and Debug Registers

In 64-bit mode, more encodings for control and debug registers are available. The REX.R bit is used to modify the ModR/M reg field when that field encodes a control or debug register (see Table 2-4). These encodings enable the processor to address CR8-CR15 and DR8- DR15. An additional control register (CR8) is defined in 64-bit mode. CR8 becomes the Task Priority Register (TPR).

In the first implementation of IA-32e mode, CR9-CR15 and DR8-DR15 are not implemented. Any attempt to access unimplemented registers results in an invalid-opcode exception (#UD).

# CHAPTER 3
# INSTRUCTION SET REFERENCE, A-M

This chapter describes the instruction set for the Intel 64 and IA-32 architectures (A-M) in IA-32e, protected, Virtual-8086, and real modes of operation. The set includes general-purpose, x87 FPU, MMX, SSE/SSE2/SSE3/SSSE3, and system instructions. See also Chapter 4, "Instruction Set Reference, N-Z," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

For each instruction, each operand combination is described. A description of the instruction and its operand, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of exceptions that can be generated are also provided.

## 3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES

This section describes the format of information contained in the instruction reference pages in this chapter. It explains notational conventions and abbreviations used in these sections.

### 3.1.1 Instruction Format

The following is an example of the format used for each instruction description in this chapter. The heading below introduces the example. The table below provides an example summary table.

#### CMC—Complement Carry Flag [this is an example]

| Opcode | Instruction | 64-bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F5 | CMC | Valid | Valid | Complement carry flag. |

### 3.1.1.1 Opcode Column in the Instruction Summary Table

The “Opcode” column in the table above shows the object code produced for each form of the instruction. When possible, codes are given as hexadecimal bytes in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.
- **/digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- **/r —** Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- **cb, cw, cd, cp, co, ct —** A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id, io —** A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords and quadwords are given with the low-order byte first.
- **+rb, +rw, +rd, +ro —** A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. See Table 3-1 for the codes. The +ro columns in the table are applicable only in 64-bit mode.
- **+i —** A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

**Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro**

| byte register | | | word register | | | dword register | | | quadword register (64-Bit Mode only) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Register | REX.B | Reg Field | Register | REX.B | Reg Field | Register | REX.B | Reg Field | Register | REX.B | Reg Field |
| AL | None | 0 | AX | None | 0 | EAX | None | 0 | RAX | None | 0 |
| CL | None | 1 | CX | None | 1 | ECX | None | 1 | RCX | None | 1 |
| DL | None | 2 | DX | None | 2 | EDX | None | 2 | RDX | None | 2 |

**Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro (Contd.)**

| byte register | | | word register | | | dword register | | | quadword register (64-Bit Mode only) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Register | REX.B | Reg Field | Register | REX.B | Reg Field | Register | REX.B | Reg Field | Register | REX.B | Reg Field |
| BL | None | 3 | BX | None | 3 | EBX | None | 3 | RBX | None | 3 |
| AH | Not encodable (N.E.) | 4 | SP | None | 4 | ESP | None | 4 | N/A | N/A | N/A |
| CH | N.E. | 5 | BP | None | 5 | EBP | None | 5 | N/A | N/A | N/A |
| DH | N.E. | 6 | SI | None | 6 | ESI | None | 6 | N/A | N/A | N/A |
| BH | N.E. | 7 | DI | None | 7 | EDI | None | 7 | N/A | N/A | N/A |
| SPL | Yes | 4 | SP | None | 4 | ESP | None | 4 | RSP | None | 4 |
| BPL | Yes | 5 | BP | None | 5 | EBP | None | 5 | RBP | None | 5 |
| SIL | Yes | 6 | SI | None | 6 | ESI | None | 6 | RSI | None | 6 |
| DIL | Yes | 7 | DI | None | 7 | EDI | None | 7 | RDI | None | 7 |
| Registers R8 - R15 (see below): Available in 64-Bit Mode Only | | | | | | | | | | | |
| R8L | Yes | 0 | R8W | Yes | 0 | R8D | Yes | 0 | R8 | Yes | 0 |
| R9L | Yes | 1 | R9W | Yes | 1 | R9D | Yes | 1 | R9 | Yes | 1 |
| R10L | Yes | 2 | R10W | Yes | 2 | R10D | Yes | 2 | R10 | Yes | 2 |
| R11L | Yes | 3 | R11W | Yes | 3 | R11D | Yes | 3 | R11 | Yes | 3 |
| R12L | Yes | 4 | R12W | Yes | 4 | R12D | Yes | 4 | R12 | Yes | 4 |
| R13L | Yes | 5 | R13W | Yes | 5 | R13D | Yes | 5 | R13 | Yes | 5 |
| R14L | Yes | 6 | R14W | Yes | 6 | R14D | Yes | 6 | R14 | Yes | 6 |
| R15L | Yes | 7 | R15W | Yes | 7 | R15D | Yes | 7 | R15 | Yes | 7 |

### 3.1.1.2 Instruction Column in the Opcode Summary Table

The “Instruction” column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

- **rel8** — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16, rel32, rel64** — A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits; the rel64 symbol applies to instructions with an operand-size attribute of 64 bits.

- **ptr16:16, ptr16:32 and ptr16:64** — A far pointer, typically to a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits; the ptr16:64 symbol is used when the operand-size attribute is 64 bits.
- **r8** — One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL; or one of the byte registers (R8L - R15L) available when using REX.R and 64-bit mode.
- **r16** — One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.
- **r32** — One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.
- **r64** — One of the quadword general-purpose registers: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15. These are available when using REX.R and 64-bit mode.
- **imm8** — An immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16** — An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.
- **imm32** — An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648 inclusive.
- **imm64** — An immediate quadword value used for instructions whose operand-size attribute is 64 bits. The value allows the use of a number between +9,223,372,036,854,775,807 and –9,223,372,036,854,775,808 inclusive.
- **r/m8** — A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL) or a byte from memory. Byte registers R8L - R15L are available using REX.R in 64-bit mode.
- **r/m16** — A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.

- **r/m32** — A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation. Doubleword registers R8D - R15D are available when using REX.R in 64-bit mode.
- **r/m64** — A quadword general-purpose register or memory operand used for instructions whose operand-size attribute is 64 bits when using REX.W. Quadword general-purpose registers are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15; these are available only in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **m** — A 16-, 32- or 64-bit operand in memory.
- **m8** — A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. In 64-bit mode, it is pointed to by the RSI or RDI registers.
- **m16** — A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m32** — A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m64** — A memory quadword operand in memory.
- **m128** — A memory double quadword operand in memory. This nomenclature is used only with SSE and SSE2 instructions.
- **m16:16, m16:32 & m16:64** — A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32, m16&64** — A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers. The m16&64 operand is used by LIDT and LGDT in 64-bit mode to provide a word with which to load the limit field, and a quadword with which to load the base field of the corresponding GDTR and IDTR registers.
- **moffs8, moffs16, moffs32, moffs64** — A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.

- **Sreg** — A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4, and GS = 5.
- **m32fp, m64fp, m80fp** — A single-precision, double-precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FPU floating-point instructions.
- **m16int, m32int, m64int** — A word, doubleword, and quadword integer (respectively) operand in memory. These symbols designate integers that are used as operands for x87 FPU integer instructions.
- **ST or ST(0)** — The top element of the FPU register stack.
- **ST(i)** — The $i^{th}$ element from the top of the FPU register stack ($i \leftarrow 0$ through 7).
- **mm** — An MMX register. The 64-bit MMX registers are: MM0 through MM7.
- **mm/m32** — The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **mm/m64** — An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **xmm** — An XMM register. The 128-bit XMM registers are: XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode.
- **xmm/m32**— An XMM register or a 32-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m64** — An XMM register or a 64-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m128** — An XMM register or a 128-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.

### 3.1.1.3 64-bit Mode Column in the Instruction Summary Table

The “64-bit Mode” column indicates whether the opcode sequence is supported in 64-bit mode. The column uses the following notation:

- **Valid** — Supported.
- **Invalid** — Not supported.
- **N.E.** — Indicates an instruction syntax is not encodable in 64-bit mode (it may represent part of a sequence of valid instructions in other modes).

- **N.P.** — Indicates the REX prefix does not affect the legacy instruction in 64-bit mode.
- **N.I.** — Indicates the opcode is treated as a new instruction in 64-bit mode.
- **N.S.** — Indicates an instruction syntax that requires an address override prefix in 64-bit mode and is not supported. Using an address override prefix in 64-bit mode may result in model-specific execution behavior.

#### 3.1.1.4 Compatibility/Legacy Mode Column in the Instruction Summary Table

The "Compatibility/Legacy Mode" column provides information on the opcode sequence in either the compatibility mode or other IA-32 modes. The column uses the following notation:

- **Valid** — Supported.
- **Invalid** — Not supported.
- **N.E.** — Indicates an Intel 64 instruction mnemonics/syntax that is not encodable; the opcode sequence is not applicable as an individual instruction in compatibility mode or IA-32 mode. The opcode may represent a valid sequence of legacy IA-32 instructions.

#### 3.1.1.5 Description Column in the Instruction Summary Table

The "Description" column briefly explains forms of the instruction.

#### 3.1.1.6 Description Section

Each instruction is then described by number of information sections. The "Description" section describes the purpose of the instructions and required operands in more detail.

#### 3.1.1.7 Operation Section

The "Operation" section contains an algorithm description (frequently written in pseudo-code) for the instruction. Algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs "(*" and "*)".
- Compound statements are enclosed in keywords, such as: IF, THEN, ELSE and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the

address contained in register SI relative to the SI register's default segment (DS) or the overridden segment.

- Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32. Parentheses around the "R" in a general-purpose register name, (R)SI, in the presence of a 64-bit register definition such as (R)SI, indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.
- Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.
- A ←B indicates that the value of B is assigned to A.
- The symbols =, ≠, >, <, ≥, and ≤are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A ←B is TRUE if the value of A is equal to B; otherwise it is FALSE.
- The expression "<< COUNT" and ">> COUNT" indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize and AddressSize** — The OperandSize identifier represents the operand-size attribute of the instruction, which is 16, 32 or 64-bits. The AddressSize identifier represents the address-size attribute, which is 16, 32 or 64-bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the MOV instruction used.

```
IF Instruction ←MOVW
    THEN OperandSize ←16;
ELSE
    IF Instruction ←MOVD
        THEN OperandSize ←32;
    ELSE
        IF Instruction ←MOVQ
            THEN OperandSize ←64;
        FI;
    FI;
FI;
```

  See "Operand-Size and Address-Size Attributes" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for guidelines on how these attributes are determined.

- **StackAddrSize** — Represents the stack address-size attribute associated with the instruction, which has a value of 16, 32 or 64-bits. See "Address-Size

Attribute for Stack" in Chapter 6, "Procedure Calls, Interrupts, and Exceptions," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

- **SRC** — Represents the source operand.
- **DEST** — Represents the destination operand.

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** — Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of –10 converts the byte from F6H to a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend(value)** — Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value –10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **SaturateSignedWordToSignedByte** — Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than –128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateSignedDwordToSignedWord** — Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateSignedWordToUnsignedByte** — Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToSignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateToSignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateToUnsignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToUnsignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value

zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).

- **LowOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.
- **HighOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.
- **Push(value)** — Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. See the "Operation" subsection of the "PUSH—Push Word, Doubleword or Quadword Onto the Stack" section in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
- **Pop()** removes the value from the top of the stack and returns it. The statement EAX ←Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word, a doubleword or a quadword depending on the operand-size attribute. See the "Operation" subsection in the "POP—Pop a Value from the Stack" section of Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
- **PopRegisterStack** — Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.
- **Switch-Tasks** — Performs a task switch.
- **Bit(BitBase, BitOffset)** — Returns the value of a bit within a bit string. The bit string is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the BitBase is a register, the BitOffset can be in the range 0 to [15, 31, 63] depending on the mode and register size. See Figure 3-1: the function Bit[RAX, 21] is illustrated.

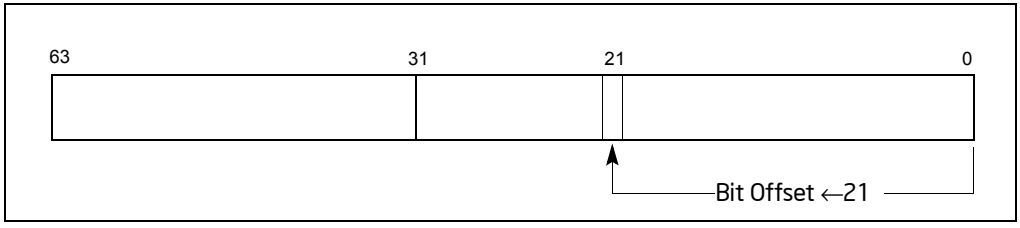



**Figure 3-1. Bit Offset for BIT[RAX, 21]**

If BitBase is a memory address, the BitOffset can range has different ranges depending on the operand size (see Table 3-2).

**Table 3-2. Range of Bit Positions Specified by Bit Offset Operands**

| Operand Size | Immediate BitOffset | Register BitOffset |
|---|---|---|
| 16 | 0 to 15 | $-2^{15}$ to $2^{15}-1$ |
| 32 | 0 to 31 | $-2^{31}$ to $2^{31}-1$ |
| 64 | 0 to 63 | $-2^{63}$ to $2^{63}-1$ |

The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)) where DIV is signed division with rounding towards negative infinity and MOD returns a positive number (see Figure 3-2).

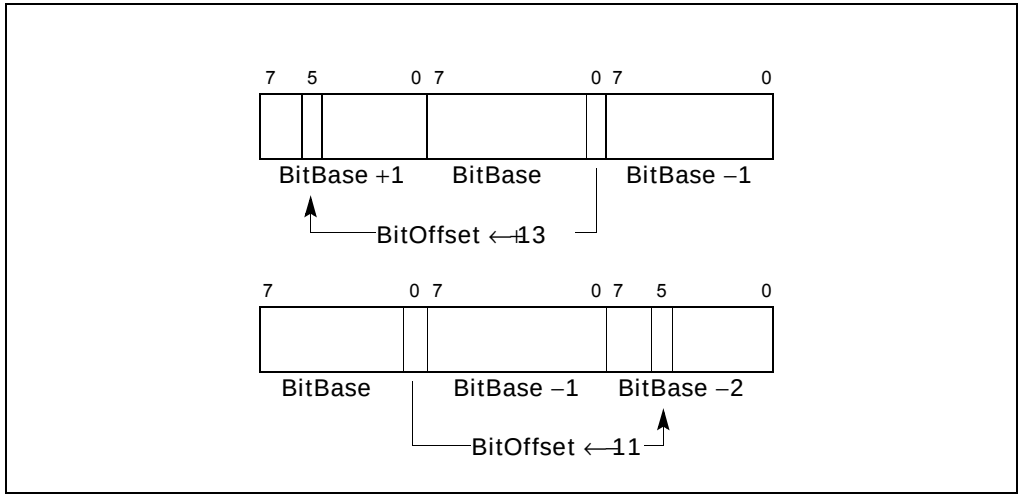



**Figure 3-2. Memory Bit Indexing**

### 3.1.1.8 Intel® C/C++ Compiler Intrinsics Equivalents Section

The Intel C/C++ compiler intrinsics equivalents are special C/C++ coding extensions that allow using the syntax of C function calls and C variables instead of hardware registers. Using these intrinsics frees programmers from having to manage registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that executable run faster.

The following sections discuss the intrinsics API and the MMX technology and SIMD floating-point intrinsics. Each intrinsic equivalent is listed with the instruction description. There may be additional intrinsics that do not have an instruction equivalent. It is strongly recommended that the reader reference the compiler documentation for the complete list of supported intrinsics.

See Appendix C, "InteL® C/C++ Compiler Intrinsics and Functional Equivalents," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for more information on using intrinsics.

## Intrinsics API

The benefit of coding with MMX technology intrinsics and the SSE/SSE2/SSE3 intrinsics is that you can use the syntax of C function calls and C variables instead of hardware registers. This frees you from managing registers and programming assembly. Further, the compiler optimizes the instruction scheduling so that your executable runs faster. For each computational and data manipulation instruction in the new instruction set, there is a corresponding C intrinsic that implements it directly. The intrinsics allow you to specify the underlying implementation (instruction selection) of an algorithm yet leave instruction scheduling and register allocation to the compiler.

## MMX™ Technology Intrinsics

The MMX technology intrinsics are based on a __m64 data type that represents the specific contents of an MMX technology register. You can specify values in bytes, short integers, 32-bit values, or a 64-bit object. The __m64 data type, however, is not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

- Use __m64 data only on the left-hand side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions ("+", ">>", and so on).
- Use __m64 objects in aggregates, such as unions to access the byte elements and structures; the address of an __m64 object may be taken.
- Use __m64 data only with the MMX technology intrinsics described in this manual and Intel® C/C++ compiler documentation.
- See:
  - http://www.intel.com/support/performancetools/
  - Appendix C, "InteL® C/C++ Compiler Intrinsics and Functional Equivalents," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for more information on using intrinsics.
  - SSE/SSE2/SSE3 Intrinsics
  - SSE/SSE2/SSE3 intrinsics all make use of the XMM registers of the Pentium III, Pentium 4, and Intel Xeon processors. There are three data types supported by these intrinsics: __m128, __m128d, and __m128i.
- The __m128 data type is used to represent the contents of an XMM register used by an SSE intrinsic. This is either four packed single-precision floating-point values or a scalar single-precision floating-point value.
- The __m128d data type holds two packed double-precision floating-point values or a scalar double-precision floating-point value.

- The __m128i data type can hold sixteen byte, eight word, or four doubleword, or two quadword integer values.

The compiler aligns __m128, __m128d, and __m128i local and global data to 16-byte boundaries on the stack. To align integer, float, or double arrays, use the declspec statement as described in Intel C/C++ compiler documentation. See http://www.intel.com/support/performancetools/.

The __m128, __m128d, and __m128i data types are not basic ANSI C data types and therefore some restrictions are placed on its usage:

- Use __m128, __m128d, and __m128i only on the left-hand side of an assignment, as a return value, or as a parameter. Do not use it in other arithmetic expressions such as "+" and ">>."
- Do not initialize __m128, __m128d, and __m128i with literals; there is no way to express 128-bit constants.
- Use __m128, __m128d, and __m128i objects in aggregates, such as unions (for example, to access the float elements) and structures. The address of these objects may be taken.
- Use __m128, __m128d, and __m128i data only with the intrinsics described in this user's guide. See Appendix C, "InteL® C/C++ Compiler Intrinsics and Functional Equivalents," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for more information on using intrinsics.

The compiler aligns __m128, __m128d, and __m128i local data to 16-byte boundaries on the stack. Global __m128 data is also aligned on 16-byte boundaries. (To align float arrays, you can use the alignment declspec described in the following section.) Because the new instruction set treats the SIMD floating-point registers in the same way whether you are using packed or scalar data, there is no __m32 data type to represent scalar data as you might expect. For scalar operations, you should use the __m128 objects and the "scalar" forms of the intrinsics; the compiler and the processor implement these operations with 32-bit memory references.

The suffixes ps and ss are used to denote "packed single" and "scalar single" precision operations. The packed floats are represented in right-to-left order, with the lowest word (right-most) being used for scalar operations: [z, y, x, w]. To explain how memory storage reflects this, consider the following example.

The operation:

```
float a[4] ←{ 1.0, 2.0, 3.0, 4.0 };
__m128 t ←_mm_load_ps(a);
```

Produces the same result as follows:

```
__m128 t ←_mm_set_ps(4.0, 3.0, 2.0, 1.0);
```

In other words:

```
t ←[ 4.0, 3.0, 2.0, 1.0 ]
```

Where the "scalar" element is 1.0.

Some intrinsics are “composites” because they require more than one instruction to implement them. You should be familiar with the hardware features provided by the SSE, SSE2, SSE3, and MMX technology when writing programs with the intrinsics.

Keep the following important issues in mind:

- Certain intrinsics, such as _mm_loadr_ps and _mm_cmpgt_ss, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful of their implementation cost.
- Data loaded or stored as __m128 objects must generally be 16-byte-aligned.
- Some intrinsics require that their argument be immediates, that is, constant integers (literals), due to the nature of the instruction.
- The result of arithmetic operations acting on two NaN (Not a Number) arguments is undefined. Therefore, floating-point operations using NaN arguments may not match the expected behavior of the corresponding assembly instructions.

For a more detailed description of each intrinsic and additional information related to its usage, refer to Intel C/C++ compiler documentation. See:

— http://www.intel.com/support/performancetools/

— Appendix C, “Intel® C/C++ Compiler Intrinsics and Functional Equivalents,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for more information on using intrinsics.

### 3.1.1.9 Flags Affected Section

The “Flags Affected” section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, “Eflags Cross-Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Non-conventional assignments are described in the “Operation” section. The values of flags listed as **undefined** may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

### 3.1.1.10 FPU Flags Affected Section

The floating-point instructions have an “FPU Flags Affected” section that describes how each instruction can affect the four condition code flags of the FPU status word.

### 3.1.1.11 Protected Mode Exceptions Section

The “Protected Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 3-3 associates each two-

letter mnemonic with the corresponding interrupt vector number and exception name. See Chapter 5, "Interrupt and Exception Handling," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

**Table 3-3. Intel 64 and IA-32 General Exceptions**

| Vector No. | Name | Source | Protected Mode[1] | Real Address Mode | Virtual 8086 Mode |
|---|---|---|---|---|---|
| 0 | #DE—Divide Error | DIV and IDIV instructions. | Yes | Yes | Yes |
| 1 | #DB—Debug | Any code or data reference. | Yes | Yes | Yes |
| 3 | #BP—Breakpoint | INT 3 instruction. | Yes | Yes | Yes |
| 4 | #OF—Overflow | INTO instruction. | Yes | Yes | Yes |
| 5 | #BR—BOUND Range Exceeded | BOUND instruction. | Yes | Yes | Yes |
| 6 | #UD—Invalid Opcode (Undefined Opcode) | UD2 instruction or reserved opcode. | Yes | Yes | Yes |
| 7 | #NM—Device Not Available (No Math Coprocessor) | Floating-point or WAIT/FWAIT instruction. | Yes | Yes | Yes |
| 8 | #DF—Double Fault | Any instruction that can generate an exception, an NMI, or an INTR. | Yes | Yes | Yes |
| 10 | #TS—Invalid TSS | Task switch or TSS access. | Yes | Reserved | Yes |
| 11 | #NP—Segment Not Present | Loading segment registers or accessing system segments. | Yes | Reserved | Yes |
| 12 | #SS—Stack Segment Fault | Stack operations and SS register loads. | Yes | Yes | Yes |
| 13 | #GP—General Protection[2] | Any memory reference and other protection checks. | Yes | Yes | Yes |
| 14 | #PF—Page Fault | Any memory reference. | Yes | Reserved | Yes |
| 16 | #MF—Floating-Point Error (Math Fault) | Floating-point or WAIT/FWAIT instruction. | Yes | Yes | Yes |
| 17 | #AC—Alignment Check | Any data reference in memory. | Yes | Reserved | Yes |

Table 3-3. Intel 64 and IA-32 General Exceptions (Contd.)

| Vector No. | Name | Source | Protected Mode[1] | Real Address Mode | Virtual 8086 Mode |
|---|---|---|---|---|---|
| 18 | #MC—Machine Check | Model dependent machine check errors. | Yes | Yes | Yes |
| 19 | #XM—SIMD Floating-Point Numeric Error | SSE/SSE2/SSE3 floating-point instructions. | Yes | Yes | Yes |

NOTES:

1. Apply to protected mode, compatibility mode, and 64-bit mode.
2. In the real-address mode, vector 13 is the segment overrun exception.

### 3.1.1.12 Real-Address Mode Exceptions Section

The “Real-Address Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in real-address mode (see Table 3-3).

### 3.1.1.13 Virtual-8086 Mode Exceptions Section

The “Virtual-8086 Mode Exceptions” section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode (see Table 3-3).

### 3.1.1.14 Floating-Point Exceptions Section

The “Floating-Point Exceptions” section lists exceptions that can occur when an x87 FPU floating-point instruction is executed. All of these exception conditions result in a floating-point error exception (#MF, vector number 16) being generated. Table 3-4 associates a one- or two-letter mnemonic with the corresponding exception name. See “Floating-Point Exception Conditions” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a detailed description of these exceptions.

Table 3-4. x87 FPU Floating-Point Exceptions

| Mnemonic | Name | Source |
|---|---|---|
| #IS<br>#IA | Floating-point invalid operation:<br>- Stack overflow or underflow<br>- Invalid arithmetic operation | <br>- x87 FPU stack overflow or underflow<br>- Invalid FPU arithmetic operation |
| #Z | Floating-point divide-by-zero | Divide-by-zero |
| #D | Floating-point denormal operand | Source operand that is a denormal number |
| #O | Floating-point numeric overflow | Overflow in result |
| #U | Floating-point numeric underflow | Underflow in result |
| #P | Floating-point inexact result (precision) | Inexact result (precision) |

### 3.1.1.15 SIMD Floating-Point Exceptions Section

The "SIMD Floating-Point Exceptions" section lists exceptions that can occur when an SSE/SSE2/SSE3 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XM, vector number 19) being generated. Table 3-5 associates a one-letter mnemonic with the corresponding exception name. For a detailed description of these exceptions, refer to "SSE and SSE2 Exceptions", in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Table 3-5. SIMD Floating-Point Exceptions

| Mnemonic | Name | Source |
|---|---|---|
| #I | Floating-point invalid operation | Invalid arithmetic operation or source operand |
| #Z | Floating-point divide-by-zero | Divide-by-zero |
| #D | Floating-point denormal operand | Source operand that is a denormal number |
| #O | Floating-point numeric overflow | Overflow in result |
| #U | Floating-point numeric underflow | Underflow in result |
| #P | Floating-point inexact result | Inexact result (precision) |

### 3.1.1.16 Compatibility Mode Exceptions Section

This section lists exception that occur within compatibility mode.

### 3.1.1.17 64-Bit Mode Exceptions Section

This section lists exception that occur within 64-bit mode.

## 3.2 INSTRUCTIONS (A-M)

The remainder of this chapter provides descriptions of Intel 64 and IA-32 instructions (A-M). See also: Chapter 4, “Instruction Set Reference, N-Z,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*.

# AAA—ASCII Adjust After Addition

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 37 | AAA | Invalid | Valid | ASCII adjust AL after addition. |

## Description

Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register increments by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

## Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
                AL ← AL +6;
                AH ← AH +1;
                AF ← 1;
                CF ← 1;
                AL ← AL AND 0FH;
            ELSE
                AF ← 0;
                CF ← 0;
                AL ← AL AND 0FH;
        FI;
FI;
```

## Flags Affected

The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.

### Protected Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

### Compatibility Mode Exceptions

Same exceptions as protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If in 64-bit mode. |

# AAD—ASCII Adjust AX Before Division

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D5 0A | AAD | Invalid | Valid | ASCII adjust AX before division. |
| D5 *ib* | (No mnemonic) | Invalid | Valid | Adjust AX before division to number base *imm8.* |

## Description

Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AX register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to (AL +(10 * AH)), and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit (base 10) number in registers AH and AL.

The generalized version of this instruction allows adjustment of two unpacked digits of any number base (see the "Operation" section below), by setting the *imm8* byte to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAD mnemonic is interpreted by all assemblers to mean adjust ASCII (base 10) values. To adjust values in another number base, the instruction must be hand coded in machine code (D5 *imm8*).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

## Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL ←AL;
        tempAH ←AH;
        AL ←(tempAL +(tempAH * imm8)) AND FFH;
        (* imm8 is set to 0AH for the AAD mnemonic.*)
        AH ←0;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

### Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register; the OF, AF, and CF flags are undefined.

### Protected Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

### Compatibility Mode Exceptions

Same exceptions as protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If in 64-bit mode. |

# AAM—ASCII Adjust AX After Multiply

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D4 0A | AAM | Invalid | Valid | ASCII adjust AX after multiply. |
| D4 *ib* | (No mnemonic) | Invalid | Valid | Adjust AX after multiply to number base *imm8*. |

## Description

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the "Operation" section below). Here, the *imm8* byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 *imm8*).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

## Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL ←AL;
        AH ←tempAL / imm8; (* imm8 is set to 0AH for the AAM mnemonic *)
        AL ←tempAL MOD imm8;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

## Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register. The OF, AF, and CF flags are undefined.

### Protected Mode Exceptions

| | |
|---|---|
| #DE | If an immediate value of 0 is used. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

### Compatibility Mode Exceptions

Same exceptions as protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If in 64-bit mode. |

# AAS—ASCII Adjust AL After Subtraction

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 3F | AAS | Invalid | Valid | ASCII adjust AL after subtraction. |

## Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register decrements by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top nibble set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

## Operation

```
IF 64-bit mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
                AL ← AL – 6;
                AH ← AH – 1;
                AF ← 1;
                CF ← 1;
                AL ← AL AND 0FH;
            ELSE
                CF ← 0;
                AF ← 0;
                AL ← AL AND 0FH;
        FI;
FI;
```

## Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

### Protected Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

### Compatibility Mode Exceptions

Same exceptions as protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If in 64-bit mode. |

## ADC—Add with Carry

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 14 *ib* | ADC AL, *imm8* | Valid | Valid | Add with carry *imm8* to AL. |
| 15 *iw* | ADC AX, *imm16* | Valid | Valid | Add with carry *imm16* to AX. |
| 15 *id* | ADC EAX, *imm32* | Valid | Valid | Add with carry *imm32* to EAX. |
| REX.W + 15 *id* | ADC RAX, *imm32* | Valid | N.E. | Add with carry *imm32 sign extended to 64-bits* to RAX. |
| 80 /2 *ib* | ADC *r/m8, imm8* | Valid | Valid | Add with carry *imm8* to *r/m8.* |
| REX + 80 /2 *ib* | ADC *r/m8**, *imm8* | Valid | N.E. | Add with carry *imm8* to *r/m8.* |
| 81 /2 *iw* | ADC *r/m16, imm16* | Valid | Valid | Add with carry *imm16* to *r/m16.* |
| 81 /2 *id* | ADC *r/m32, imm32* | Valid | Valid | Add with CF *imm32* to *r/m32.* |
| REX.W + 81 /2 *id* | ADC *r/m64, imm32* | Valid | N.E. | Add with CF *imm32* sign extended to 64-bits to *r/m64.* |
| 83 /2 *ib* | ADC *r/m16, imm8* | Valid | Valid | Add with CF sign-extended *imm8* to *r/m16.* |
| 83 /2 *ib* | ADC *r/m32, imm8* | Valid | Valid | Add with CF sign-extended *imm8* into *r/m32.* |
| REX.W + 83 /2 *ib* | ADC *r/m64, imm8* | Valid | N.E. | Add with CF sign-extended *imm8* into *r/m64.* |
| 10 /*r* | ADC *r/m8, r8* | Valid | Valid | Add with carry byte register to *r/m8.* |
| REX + 10 /*r* | ADC *r/m8**, *r8** | Valid | N.E. | Add with carry byte register to *r/m64.* |
| 11 /*r* | ADC *r/m16, r16* | Valid | Valid | Add with carry *r16* to *r/m16.* |
| 11 /*r* | ADC *r/m32, r32* | Valid | Valid | Add with CF *r32* to *r/m32.* |
| REX.W + 11 /*r* | ADC *r/m64, r64* | Valid | N.E. | Add with CF *r64* to *r/m64.* |
| 12 /*r* | ADC *r8, r/m8* | Valid | Valid | Add with carry *r/m8* to byte register. |
| REX + 12 /*r* | ADC *r8**, *r/m8** | Valid | N.E. | Add with carry *r/m64* to byte register. |
| 13 /*r* | ADC *r16, r/m16* | Valid | Valid | Add with carry *r/m16* to *r16.* |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 13 /r | ADC *r32, r/m32* | Valid | Valid | Add with CF *r/m32* to *r32.* |
| REX.W + 13 /r | ADC *r64, r/m64* | Valid | N.E. | Add with CF *r/m64* to *r64.* |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
DEST ←DEST +SRC +CF;
```

## Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## ADD—Add

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 04 *ib* | ADD AL, *imm8* | Valid | Valid | Add *imm8* to AL. |
| 05 *iw* | ADD AX, *imm16* | Valid | Valid | Add *imm16* to AX. |
| 05 *id* | ADD EAX, *imm32* | Valid | Valid | Add *imm32* to EAX. |
| REX.W + 05 *id* | ADD RAX, *imm32* | Valid | N.E. | Add *imm32 sign-extended to 64-bits* to RAX. |
| 80 /0 *ib* | ADD *r/m8, imm8* | Valid | Valid | Add *imm8* to *r/m8.* |
| REX + 80 /0 *ib* | ADD *r/m8**, *imm8* | Valid | N.E. | Add *sign*-extended *imm8* to *r/m64.* |
| 81 /0 *iw* | ADD *r/m16, imm16* | Valid | Valid | Add *imm16* to *r/m16.* |
| 81 /0 *id* | ADD *r/m32, imm32* | Valid | Valid | Add *imm32* to *r/m32.* |
| REX.W + 81 /0 *id* | ADD *r/m64, imm32* | Valid | N.E. | Add *imm32 sign-extended to 64-bits* to *r/m64.* |
| 83 /0 *ib* | ADD *r/m16, imm8* | Valid | Valid | Add *sign*-extended *imm8* to *r/m16.* |
| 83 /0 *ib* | ADD *r/m32, imm8* | Valid | Valid | Add *sign*-extended *imm8* to *r/m32.* |
| REX.W + 83 /0 *ib* | ADD *r/m64, imm8* | Valid | N.E. | Add *sign*-extended *imm8* to *r/m64.* |
| 00 /*r* | ADD *r/m8, r8* | Valid | Valid | Add *r8* to *r/m8.* |
| REX + 00 /*r* | ADD *r/m8**, *r8** | Valid | N.E. | Add *r8* to *r/m8.* |
| 01 /*r* | ADD *r/m16, r16* | Valid | Valid | Add *r16* to *r/m16.* |
| 01 /*r* | ADD *r/m32, r32* | Valid | Valid | Add r32 to *r/m32.* |
| REX.W + 01 /*r* | ADD *r/m64, r64* | Valid | N.E. | Add r64 to *r/m64.* |
| 02 /*r* | ADD *r8, r/m8* | Valid | Valid | Add *r/m8* to *r8.* |
| REX + 02 /*r* | ADD *r8**, *r/m8** | Valid | N.E. | Add *r/m8* to *r8.* |
| 03 /*r* | ADD *r16, r/m16* | Valid | Valid | Add *r/m16* to *r16.* |
| 03 /*r* | ADD *r32, r/m32* | Valid | Valid | Add *r/m32* to *r32.* |
| REX.W + 03 /*r* | ADD *r64, r/m64* | Valid | N.E. | Add *r/m64* to *r64.* |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
DEST ←DEST +SRC;
```

## Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

# ADDPD—Add Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 58 /r | ADDPD *xmm1, xmm2/m128* | Valid | Valid | Add packed double-precision floating-point values from *xmm2/m128* to *xmm1*. |

## Description

Performs a SIMD add of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←DEST[63:0] +SRC[63:0];
DEST[127:64] ←DEST[127:64] +SRC[127:64];
```

## Intel C/C++Compiler Intrinsic Equivalent

ADDPD __m128d _mm_add_pd (m128d a, m128d b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CRO.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CRO.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment.<br>If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form.<br>If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# ADDPS—Add Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 58 /r | ADDPS *xmm1, xmm2/m128* | Valid | Valid | Add packed single-precision floating-point values from *xmm2/m128* to *xmm1*. |

## Description

Performs a SIMD add of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of SIMD single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←DEST[31:0] +SRC[31:0];
DEST[63:32] ←DEST[63:32] +SRC[63:32];
DEST[95:64] ←DEST[95:64] +SRC[95:64];
DEST[127:96] ←DEST[127:96] +SRC[127:96];
```

## Intel C/C++ Compiler Intrinsic Equivalent

ADDPS __m128 _mm_add_ps(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

# ADDSD—Add Scalar Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 58 /r | ADDSD *xmm1, xmm2/m64* | Valid | Valid | Add the low double-precision floating-point value from *xmm2/m64* to *xmm1*. |

## Description

Adds the low double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the double-precision floating-point result in the destination operand.

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a scalar double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←DEST[63:0] +SRC[63:0];
(* DEST[127:64] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

ADDSD __m128d _mm_add_sd (m128d a, m128d b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CRO.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# ADDSS—Add Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 58 /r | ADDSS *xmm1, xmm2/m32* | Valid | Valid | Add the low single-precision floating-point value from *xmm2/m32* to *xmm1*. |

## Description

Adds the low single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the single-precision floating-point result in the destination operand.

The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a scalar single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←DEST[31:0] +SRC[31:0];
(* DEST[127:32] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

ADDSS __m128 _mm_add_ss(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CRO.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CRO.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CRO.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# ADDSUBPD—Packed Double-FP Add/Subtract

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F D0 /r | ADDSUBPD *xmm1, xmm2/m128* | Valid | Valid | Add/subtract double-precision floating-point values from *xmm2/m128* to *xmm1*. |

## Description

Adds the double-precision floating-point values in the high quadword of the source and destination operands and stores the result in the high quadword of the destination operand.

Subtracts the double-precision floating-point value in the low quadword of the source operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand. See Figure 3-3.

The source operand can be a 128-bit memory location or an XMM register. The destination operand is an XMM register.

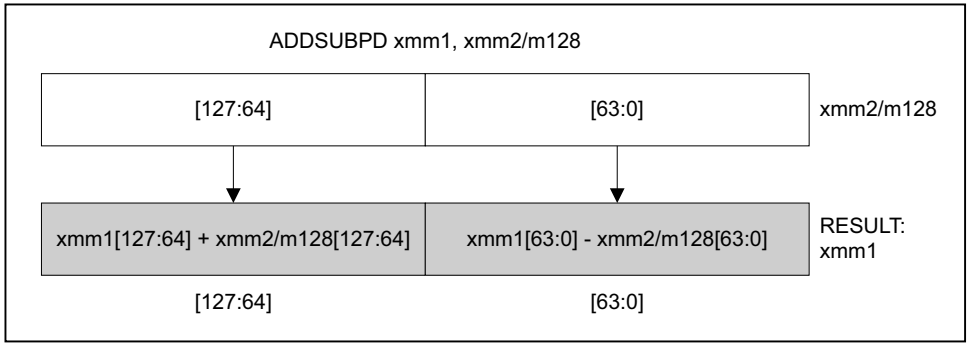



Figure 3-3. ADDSUBPD—Packed Double-FP Add/Subtract

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
xmm1[63:0] = xmm1[63:0] –xmm2/m128[63:0];
xmm1[127:64] = xmm1[127:64] +xmm2/m128[127:64];
```

## Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPD __m128d _mm_addsub_pd(__m128d a, __m128d b)

## Exceptions

When the source operand is a memory operand, it must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception, CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If CR0.EM is 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

## Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If TS bit in CR0 is 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception, CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception, CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:ECX.SSE3[bit 0] = 0.<br>If the LOCK prefix is used. |

# ADDSUBPS—Packed Single-FP Add/Subtract

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F D0 /r | ADDSUBPS *xmm1, xmm2/m128* | Valid | Valid | Add/subtract single-precision floating-point values from *xmm2/m128* to *xmm1*. |

## Description

Adds odd-numbered single-precision floating-point values of the source operand (second operand) with the corresponding single-precision floating-point values from the destination operand (first operand); stores the result in the odd-numbered values of the destination operand.

Subtracts the even-numbered single-precision floating-point values in the source operand from the corresponding single-precision floating values in the destination operand; stores the result into the even-numbered values of the destination operand.

The source operand can be a 128-bit memory location or an XMM register. The destination operand is an XMM register. See Figure 3-4.

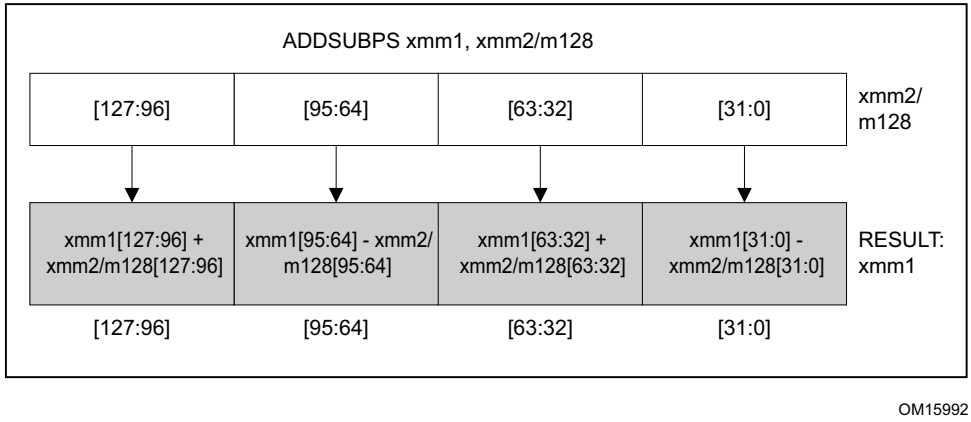



Figure 3-4. ADDSUBPS—Packed Single-FP Add/Subtract

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
xmm1[31:0] = xmm1[31:0] –xmm2/m128[31:0];
xmm1[63:32] = xmm1[63:32]  +xmm2/m128[63:32];
xmm1[95:64] = xmm1[95:64]  –xmm2/m128[95:64];
xmm1[127:96] = xmm1[127:96] +xmm2/m128[127:96];
```

## Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPS __m128 _mm_addsub_ps(__m128 a, __m128 b)

## Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception, CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

## Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #XM | For an unmasked Streaming SIMD Extensions numeric exception, CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception, CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:ECX.SSE3[bit 0] = 0.<br>If the LOCK prefix is used. |

## AND—Logical AND

| Opcode | Instruction | 64-Bit Mode | Comp/Leg Mode | Description |
|---|---|---|---|---|
| 24 *ib* | AND AL, *imm8* | Valid | Valid | AL AND *imm8.* |
| 25 *iw* | AND AX, *imm16* | Valid | Valid | AX AND i*mm16.* |
| 25 *id* | AND EAX, *imm32* | Valid | Valid | EAX AND *imm32.* |
| REX.W + 25 *id* | AND RAX, *imm32* | Valid | N.E. | RAX AND *imm32 sign-extended to 64-bits.* |
| 80 /4 *ib* | AND *r/m8, imm8* | Valid | Valid | *r/m8* AND *imm8.* |
| REX + 80 /4 *ib* | AND *r/m8**, *imm8* | Valid | N.E. | *r/m64* AND *imm8 (sign-extended).* |
| 81 /4 *iw* | AND *r/m16, imm16* | Valid | Valid | *r/m16* AND *imm16.* |
| 81 /4 *id* | AND *r/m32, imm32* | Valid | Valid | *r/m32* AND *imm32.* |
| REX.W + 81 /4 *id* | AND *r/m64, imm32* | Valid | N.E. | *r/m64* AND *imm32 sign extended to 64-bits.* |
| 83 /4 *ib* | AND *r/m16, imm8* | Valid | Valid | *r/m16* AND *imm8 (sign-extended).* |
| 83 /4 *ib* | AND *r/m32, imm8* | Valid | Valid | *r/m32* AND *imm8 (sign-extended).* |
| REX.W + 83 /4 *ib* | AND *r/m64, imm8* | Valid | N.E. | *r/m64* AND *imm8 (sign-extended).* |
| 20 /*r* | AND *r/m8, r8* | Valid | Valid | *r/m8* AND *r8.* |
| REX + 20 /*r* | AND *r/m8**, *r8** | Valid | N.E. | *r/m64* AND *r8 (sign-extended).* |
| 21 /*r* | AND *r/m16, r16* | Valid | Valid | *r/m16* AND *r16.* |
| 21 /*r* | AND *r/m32, r32* | Valid | Valid | *r/m32* AND *r32.* |
| REX.W + 21 /*r* | AND *r/m64, r64* | Valid | N.E. | *r/m64* AND *r32.* |
| 22 /*r* | AND *r8, r/m8* | Valid | Valid | *r8* AND *r/m8.* |
| REX + 22 /*r* | AND *r8**, *r/m8** | Valid | N.E. | *r/m64* AND *r8 (sign-extended).* |
| 23 /*r* | AND *r16, r/m16* | Valid | Valid | *r16* AND *r/m16.* |
| 23 /*r* | AND *r32, r/m32* | Valid | Valid | *r32* AND *r/m32.* |
| REX.W + 23 /*r* | AND *r64, r/m64* | Valid | N.E. | *r64* AND *r/m64.* |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow the it to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
DEST ←DEST AND SRC;
```

## Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination operand points to a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |

| | |
|---|---|
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

# ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 54 /r | ANDPD *xmm1, xmm2/m128* | Valid | Valid | Bitwise logical AND of *xmm2/m128* and *xmm1*. |

## Description

Performs a bitwise logical AND of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[127:0] ←DEST[127:0] BitwiseAND SRC[127:0];
```

## Intel C/C++ Compiler Intrinsic Equivalent

ANDPD __m128d _mm_and_pd(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 54 /r | ANDPS *xmm1, xmm2/m128* | Valid | Valid | Bitwise logical AND of *xmm2/m128* and *xmm1*. |

## Description

Performs a bitwise logical AND of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

DEST[127:0] ←DEST[127:0] BitwiseAND SRC[127:0];

## Intel C/C++Compiler Intrinsic Equivalent

ANDPS __m128 _mm_and_ps(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

# ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 55 /r | ANDNPD *xmm1, xmm2/m128* | Valid | Valid | Bitwise logical AND NOT of *xmm2/m128* and *xmm1*. |

## Description

Inverts the bits of the two packed double-precision floating-point values in the destination operand (first operand), performs a bitwise logical AND of the two packed double-precision floating-point values in the source operand (second operand) and the temporary inverted result, and stores the result in the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

DEST[127:0] ←(NOT(DEST[127:0])) BitwiseAND (SRC[127:0]);

## Intel C/C++ Compiler Intrinsic Equivalent

ANDNPD __m128d _mm_andnot_pd(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 55 /r | ANDNPS *xmm1, xmm2/m128* | Valid | Valid | Bitwise logical AND NOT of *xmm2/m128* and *xmm1*. |

## Description

Inverts the bits of the four packed single-precision floating-point values in the destination operand (first operand), performs a bitwise logical AND of the four packed single-precision floating-point values in the source operand (second operand) and the temporary inverted result, and stores the result in the destination operand.

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

DEST[127:0] ←(NOT(DEST[127:0])) BitwiseAND (SRC[127:0]);

## Intel C/C++Compiler Intrinsic Equivalent

ANDNPS __m128 _mm_andnot_ps(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

# ARPL—Adjust RPL Field of Segment Selector

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 63 /r | ARPL *r/m16, r16* | N. E. | Valid | Adjust RPL of *r/m16* to not less than RPL of *r16.* |

## Description

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program's code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then insures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program (the segment selector for the application program's code segment can be read from the stack following a procedure call).

This instruction executes as described in compatibility mode and legacy mode. It is not encodable in 64-bit mode.

See "Checking Caller Access Privileges" in Chapter 3, "Protected-Mode Memory Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information about the use of this instruction.

## Operation

```
IF 64-BIT MODE
    THEN
        See MOVSXD;
    ELSE
        IF DEST[RPL) < SRC[RPL)
            THEN
                ZF ←1;
                DEST[RPL) ←SRC[RPL);
```

```
            ELSE
                ZF ←0;
        FI;
FI;
```

## Flags Affected

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, it is set to 0.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | The ARPL instruction is not recognized in real-address mode. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | The ARPL instruction is not recognized in virtual-8086 mode. |
| | If the LOCK prefix is used. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Not applicable.

# BOUND—Check Array Index Against Bounds

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 62 /r | BOUND *r16, m16&16* | Invalid | Valid | Check if *r16* (array index) is within bounds specified by *m16&16.* |
| 62 /r | BOUND *r32, m32&32* | Invalid | Valid | Check if *r32* (array index) is within bounds specified by *m16&16.* |

## Description

BOUND determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. When this exception is generated, the saved return instruction pointer points to the BOUND instruction.

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

## Operation

```
IF 64bit Mode
    THEN
        #UD;
    ELSE
        IF (ArrayIndex < LowerBound OR ArrayIndex > UpperBound)
        (* Below lower bound or above upper bound *)
            THEN #BR; FI;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #BR | If the bounds test fails. |
| #UD | If second operand is not a memory location. |
| | If the LOCK prefix is used. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #BR | If the bounds test fails. |
| #UD | If second operand is not a memory location. |
| | If the LOCK prefix is used. |
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #BR | If the bounds test fails. |
| #UD | If second operand is not a memory location. |
| | If the LOCK prefix is used. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If in 64-bit mode. |

# BSF—Bit Scan Forward

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F BC /r | BSF *r16, r/m16* | Valid | Valid | Bit scan forward on *r/m16.* |
| 0F BC /r | BSF *r32, r/m32* | Valid | Valid | Bit scan forward on *r/m32.* |
| REX.W + 0F BC | BSF *r64, r/m64* | Valid | N.E. | Bit scan forward on *r/m64.* |

## Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is 0, the content of the destination operand is undefined.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF SRC = 0
    THEN
        ZF ←1;
        DEST is undefined;
    ELSE
        ZF ←0;
        temp ←0;
        WHILE Bit(SRC, temp) = 0
        DO
            temp ←temp +1;
            DEST ←temp;
        OD;
FI;
```

## Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# BSR—Bit Scan Reverse

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F BD /r | BSR *r16, r/m16* | Valid | Valid | Bit scan reverse on *r/m16.* |
| 0F BD /r | BSR *r32, r/m32* | Valid | Valid | Bit scan reverse on *r/m32.* |
| REX.W + 0F BD | BSR *r64, r/m64* | Valid | N.E. | Bit scan reverse on *r/m64.* |

## Description

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is 0, the content of the destination operand is undefined.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF SRC = 0
    THEN
        ZF ←1;
        DEST is undefined;
    ELSE
        ZF ←0;
        temp ←OperandSize – 1;
        WHILE Bit(SRC, temp) = 0
        DO
            temp ←temp –1;
            DEST ←temp;
        OD;
FI;
```

## Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# BSWAP—Byte Swap

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F C8+*rd* | BSWAP *r32* | Valid* | Valid | Reverses the byte order of a 32-bit register. |
| REX.W + 0F C8+*rd* | BSWAP *r64* | Valid | N.E. | Reverses the byte order of a 64-bit register. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Reverses the byte order of a 32-bit or 64-bit (destination) register. This instruction is provided for converting little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## IA-32 Architecture Legacy Compatibility

The BSWAP instruction is not supported on IA-32 processors earlier than the Intel486™ processor family. For compatibility with this instruction, software should include functionally equivalent code for execution on Intel processors earlier than the Intel486 processor family.

## Operation

```
TEMP ←DEST
IF 64-bit mode AND OperandSize = 64
    THEN
        DEST[7:0] ←TEMP[63:56];
        DEST[15:8] ←TEMP[55:48];
        DEST[23:16] ←TEMP[47:40];
        DEST[31:24] ←TEMP[39:32];
        DEST[39:32] ←TEMP[31:24];
        DEST[47:40] ←TEMP[23:16];
        DEST[55:48] ←TEMP[15:8];
        DEST[63:56] ←TEMP[7:0];
    ELSE
        DEST[7:0] ←TEMP[31:24];
```

```
        DEST[15:8] ←TEMP[23:16];
        DEST[23:16] ←TEMP[15:8];
        DEST[31:24] ←TEMP[7:0];
FI;
```

## Flags Affected

None.

## Exceptions (All Operating Modes)

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

# BT—Bit Test

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F A3 | BT *r/m16, r16* | Valid | Valid | Store selected bit in CF flag. |
| 0F A3 | BT *r/m32, r32* | Valid | Valid | Store selected bit in CF flag. |
| REX.W + 0F A3 | BT *r/m64, r64* | Valid | N.E. | Store selected bit in CF flag. |
| 0F BA /4 *ib* | BT *r/m16, imm8* | Valid | Valid | Store selected bit in CF flag. |
| 0F BA /4 *ib* | BT *r/m32, imm8* | Valid | Valid | Store selected bit in CF flag. |
| REX.W + 0F BA /4 *ib* | BT *r/m64, imm8* | Valid | N.E. | Store selected bit in CF flag. |

## Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset (specified by the second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode).
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

Effective Address +(4 * (BitOffset DIV 32))

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

Effective Address + (2 * (BitOffset DIV 16))

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
CF ← Bit(BitBase, BitOffset);
```

## Flags Affected

The CF flag contains the value of the selected bit. The OF, SF, ZF, AF, and PF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## BTC—Bit Test and Complement

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F BB | BTC *r/m16, r16* | Valid | Valid | Store selected bit in CF flag and complement. |
| 0F BB | BTC *r/m32, r32* | Valid | Valid | Store selected bit in CF flag and complement. |
| REX.W + 0F BB | BTC *r/m64, r64* | Valid | N.E. | Store selected bit in CF flag and complement. |
| 0F BA /7 *ib* | BTC *r/m16, imm8* | Valid | Valid | Store selected bit in CF flag and complement. |
| 0F BA /7 *ib* | BTC *r/m32, imm8* | Valid | Valid | Store selected bit in CF flag and complement. |
| REX.W + 0F BA /7 *ib* | BTC *r/m64, imm8* | Valid | N.E. | Store selected bit in CF flag and complement. |

### Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX

prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
CF ←Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ←NOT Bit(BitBase, BitOffset);
```

## Flags Affected

The CF flag contains the value of the selected bit before it is complemented. The OF, SF, ZF, AF, and PF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination operand points to a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

# BTR—Bit Test and Reset

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F B3 | BTR *r/m16, r16* | Valid | Valid | Store selected bit in CF flag and clear. |
| 0F B3 | BTR *r/m32, r32* | Valid | Valid | Store selected bit in CF flag and clear. |
| REX.W + 0F B3 | BTR *r/m64, r64* | Valid | N.E. | Store selected bit in CF flag and clear. |
| 0F BA /6 *ib* | BTR *r/m16, imm8* | Valid | Valid | Store selected bit in CF flag and clear. |
| 0F BA /6 *ib* | BTR *r/m32, imm8* | Valid | Valid | Store selected bit in CF flag and clear. |
| REX.W + 0F BA /6 *ib* | BTR *r/m64, imm8* | Valid | N.E. | Store selected bit in CF flag and clear. |

## DESCRIPTION

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX

prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
CF ←Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ←0;
```

## Flags Affected

The CF flag contains the value of the selected bit before it is cleared. The OF, SF, ZF, AF, and PF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination operand points to a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

# BTS—Bit Test and Set

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F AB | BTS *r/m16, r16* | Valid | Valid | Store selected bit in CF flag and set. |
| 0F AB | BTS *r/m32, r32* | Valid | Valid | Store selected bit in CF flag and set. |
| REX.W + 0F AB | BTS *r/m64, r64* | Valid | N.E. | Store selected bit in CF flag and set. |
| 0F BA /5 *ib* | BTS *r/m16, imm8* | Valid | Valid | Store selected bit in CF flag and set. |
| 0F BA /5 *ib* | BTS *r/m32, imm8* | Valid | Valid | Store selected bit in CF flag and set. |
| REX.W + 0F BA /5 *ib* | BTS *r/m64, imm8* | Valid | N.E. | Store selected bit in CF flag and set. |

## Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-10.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX

prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
CF ←Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ←1;
```

## Flags Affected

The CF flag contains the value of the selected bit before it is set. The OF, SF, ZF, AF, and PF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination operand points to a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

## CALL—Call Procedure

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| E8 *cw* | CALL *rel16* | N.S. | Valid | Call near, relative, displacement relative to next instruction. |
| E8 *cd* | CALL *rel32* | Valid | Valid | Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode. |
| FF /2 | CALL *r/m16* | N.E. | Valid | Call near, absolute indirect, address given in *r/m16.* |
| FF /2 | CALL *r/m32* | N.E. | Valid | Call near, absolute indirect, address given in *r/m32.* |
| FF /2 | CALL *r/m64* | Valid | N.E. | Call near, absolute indirect, address given in *r/m*64. |
| 9A *cd* | CALL *ptr16:16* | Invalid | Valid | Call far, absolute, address given in operand. |
| 9A *cp* | CALL *ptr16:32* | Invalid | Valid | Call far, absolute, address given in operand. |
| FF /3 | CALL *m16:16* | Valid | Valid | Call far, absolute indirect address given in *m16:16.*<br>In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction. |
| FF /3 | CALL *m16:32* | Valid | Valid | In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction. |
| REX.W + FF /3 | CALL *m16:64* | Valid | N.E. | In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction. |

### Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of

the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls:

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See "Calling Procedures Using Call and RET" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 6, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the CALL instruction.

**Near Call.** When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16*, *r/m32, or r/m64*). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32,

or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

**Far Calls in Real-Address or Virtual-8086 Mode.** When executing a far call in real-address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a "far branch" to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

**Far Calls in Protected Mode.** When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level
- Far call to a different privilege level (inter-privilege level call)
- Task switch (far call to another task)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand- size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the

segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an optional set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task's code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 6, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on the mechanics of a task switch.

When you execute at task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

**Mixing 16-Bit and 32-Bit Calls.** When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 16, "Mixing 16-Bit and 32-Bit Code," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information.

**Far Calls in Compatibility Mode.** When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction can not be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor

pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

**Near/(Far) Calls in 64-bit Mode.** When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (*m16:16, m16:32* or *m16:64*). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (*m16:16, m16:32* or *m16:64*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

## Operation

```
IF near call
    THEN IF near relative call
        THEN
            IF OperandSize = 64
                THEN
                    tempDEST ←SignExtend(DEST); (* DEST is rel32 *)
                    tempRIP ←RIP +tempDEST;
                    IF stack not large enough for a 8-byte return address
                        THEN #SS(0); FI;
                    Push(RIP);
                    RIP ←tempRIP;
            FI;
            IF OperandSize = 32
                THEN
                    tempEIP ←EIP +DEST; (* DEST is rel32 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 4-byte return address
                        THEN #SS(0); FI;
                    Push(EIP);
                    EIP ←tempEIP;
            FI;
            IF OperandSize = 16
                THEN
                    tempEIP ←(EIP +DEST) AND 0000FFFFH; (* DEST is rel16 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 2-byte return address
                        THEN #SS(0); FI;
                    Push(IP);
                    EIP ←tempEIP;
            FI;
        ELSE (* Near absolute call *)
            IF OperandSize = 64
```

```
                THEN
                    tempRIP ←DEST; (* DEST is r/m64 *)
                    IF stack not large enough for a 8-byte return address
                        THEN #SS(0); FI;
                    Push(RIP);
                    RIP ←tempRIP;
            FI;
            IF OperandSize = 32
                THEN
                    tempEIP ←DEST; (* DEST is r/m32 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 4-byte return address
                        THEN #SS(0); FI;
                    Push(EIP);
                    EIP ←tempEIP;
            FI;
            IF OperandSize = 16
                THEN
                    tempEIP ←DEST AND 0000FFFFH; (* DEST is r/m16 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 2-byte return address
                        THEN #SS(0); FI;
                    Push(IP);
                    EIP ←tempEIP;
            FI;
    FI;rel/abs
FI; near

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        IF OperandSize = 32
            THEN
                IF stack not large enough for a 6-byte return address
                    THEN #SS(0); FI;
                IF DEST[31:16] is not zero THEN #GP(0); FI;
                Push(CS); (* Padded with 16 high-order bits *)
                Push(EIP);
                CS ←DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
                EIP ←DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                IF stack not large enough for a 4-byte return address
                    THEN #SS(0); FI;
                Push(CS);
```

```
                Push(IP);
                CS ←DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                EIP ←DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
        FI;
FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
    THEN
        IF segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new code segment selector); FI;
        Read type and access rights of selected segment descriptor;
        IF IA32_EFER.LMA = 0
            THEN
                IF segment type is not a conforming or nonconforming code segment, call
                gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment or
                64-bit call gate,
                    THEN #GP(segment selector); FI;
        FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
FI;

CONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF DPL > CPL
        THEN #GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP ←DEST(Offset);
    IF OperandSize = 16
        THEN
```

```
            tempEIP ←tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
    segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* Padded with 16 high-order bits *)
            Push(EIP);
            CS ←DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) ←CPL;
            EIP ←tempEIP;
        ELSE
            IF OperandSize = 16
                THEN
                    Push(CS);
                    Push(IP);
                    CS ←DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ←CPL;
                    EIP ←tempEIP;
                ELSE (* OperandSize = 64 *)
                    Push(CS); (* Padded with 48 high-order bits *)
                    Push(RIP);
                    CS ←DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ←CPL;
                    RIP ←tempEIP;
            FI;
    FI;
END;

NONCONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) or (DPL ≠ CPL)
        THEN #GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
```

```
    tempEIP ←DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ←tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
    segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* Padded with 16 high-order bits *)
            Push(EIP);
            CS ←DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) ←CPL;
            EIP ←tempEIP;
        ELSE
            IF OperandSize = 16
                THEN
                    Push(CS);
                    Push(IP);
                    CS ←DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ←CPL;
                    EIP ←tempEIP;
                ELSE (* OperandSize = 64 *)
                    Push(CS); (* Padded with 48 high-order bits *)
                    Push(RIP);
                    CS ←DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ←CPL;
                    RIP ←tempEIP;
            FI;
    FI;
END;

CALL-GATE:
    IF call gate (DPL < CPL) or (RPL > DPL)
        THEN #GP(call gate selector); FI;
    IF call gate not present
        THEN #NP(call gate selector); FI;
    IF call gate code-segment selector is NULL
        THEN #GP(0); FI;
    IF call gate code-segment selector index is outside descriptor table limits
```

```
        THEN #GP(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
    or code-segment segment descriptor DPL > CPL
        THEN #GP(code segment selector); FI;
    IF IA32_EFER.LMA = 1 AND (code-segment segment descriptor is
    not a 64-bit code segment or code-segment descriptor has both L-Bit and D-bit set)
        THEN #GP(code segment selector); FI;
    IF code segment not present
        THEN #NP(new code segment selector); FI;
    IF code segment is non-conforming and DPL < CPL
        THEN go to MORE-PRIVILEGE;
        ELSE go to SAME-PRIVILEGE;
    FI;
END;

MORE-PRIVILEGE:
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ←new code segment (DPL * 8) +4;
            IF (TSSstackAddress +7) > TSS limit
                THEN #TS(current TSS selector); FI;
            newSS ←TSSstackAddress +4;
            newESP ←stack address;
        ELSE
            IF current TSS is 16-bit TSS
                THEN
                    TSSstackAddress ←new code segment (DPL * 4) +2;
                    IF (TSSstackAddress +4) > TSS limit
                        THEN #TS(current TSS selector); FI;
                    newESP ←TSSstackAddress;
                    newSS ←TSSstackAddress +2;
                ELSE (* TSS is 64-bit *)
                    TSSstackAddress ←new code segment (DPL * 8) +4;
                    IF (TSSstackAddress +8) > TSS limit
                        THEN #TS(current TSS selector); FI;
                    newESP ←TSSstackAddress;
                    newSS ←NULL;
            FI;
    FI;
    IF IA32_EFER.LMA = 0 and stack segment selector = NULL
        THEN #TS(stack segment selector); FI;
    Read code segment descriptor;
    IF IA32_EFER.LMA = 0 and (stack segment selector's RPL ≠ DPL of code segment
```

```
or stack segment DPL ≠ DPL of code segment or stack segment is not a
writable data segment)
    THEN #TS(SS selector); FI
IF IA32_EFER.LMA = 0 and stack segment not present
    THEN #SS(SS selector); FI;
IF CallGateSize = 32
    THEN
        IF stack does not have room for parameters plus 16 bytes
            THEN #SS(SS selector); FI;
        IF CallGate(InstructionPointer) not within code segment limit
            THEN #GP(0); FI;
        SS ←newSS;
        (* Segment descriptor information also loaded *)
        ESP ←newESP;
        CS:EIP ←CallGate(CS:InstructionPointer);
        (* Segment descriptor information also loaded *)
        Push(oldSS:oldESP); (* From calling procedure *)
        temp ←parameter count from call gate, masked to 5 bits;
        Push(parameters from calling procedure's stack, temp)
        Push(oldCS:oldEIP); (* Return address to calling procedure *)
    ELSE
        IF CallGateSize = 16
            THEN
                IF stack does not have room for parameters plus 8 bytes
                    THEN #SS(SS selector); FI;
                IF (CallGate(InstructionPointer) AND FFFFH) not in code segment limit
                    THEN #GP(0); FI;
                SS ←newSS;
                (* Segment descriptor information also loaded *)
                ESP ←newESP;
                CS:IP ←CallGate(CS:InstructionPointer);
                (* Segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* From calling procedure *)
                temp ←parameter count from call gate, masked to 5 bits;
                Push(parameters from calling procedure's stack, temp)
                Push(oldCS:oldEIP); (* Return address to calling procedure *)
            ELSE (* CallGateSize = 64 *)
                IF pushing 32 bytes on the stack touches non-canonical addresses
                    THEN #SS(SS selector); FI;
                IF (CallGate(InstructionPointer) is non-canonical)
                    THEN #GP(0); FI;
                SS ←newSS; (* New SS is NULL)
                RSP ←newESP;
                CS:IP ←CallGate(CS:InstructionPointer);
```

```
                    (* Segment descriptor information also loaded *)
                    Push(oldSS:oldESP); (* From calling procedure *)
                    Push(oldCS:oldEIP); (* Return address to calling procedure *)
            FI;
    FI;
    CPL ←CodeSegment(DPL)
    CS(RPL) ←CPL
END;

SAME-PRIVILEGE:
    IF CallGateSize = 32
        THEN
            IF stack does not have room for 8 bytes
                THEN #SS(0); FI;
            IF CallGate(InstructionPointer) not within code segment limit
                THEN #GP(0); FI;
            CS:EIP ←CallGate(CS:EIP) (* Segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE
            If CallGateSize = 16
                THEN
                    IF stack does not have room for 4 bytes
                        THEN #SS(0); FI;
                    IF CallGate(InstructionPointer) not within code segment limit
                        THEN #GP(0); FI;
                    CS:IP ←CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldIP); (* Return address to calling procedure *)
                ELSE (* CallGateSize = 64)
                    IF pushing 16 bytes on the stack touches non-canonical addresses
                        THEN #SS(0); FI;
                    IF RIP non-canonical
                        THEN #GP(0); FI;
                    CS:IP ←CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldIP); (* Return address to calling procedure *)
            FI;
    FI;
    CS(RPL) ←CPL
END;

TASK-GATE:
    IF task gate DPL < CPL or RPL
```

```
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
        THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;
    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector);  FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
```

## Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the target offset in destination operand is beyond the new code segment limit. |
| | If the segment selector in the destination operand is NULL. |
| | If the code segment selector in the gate is NULL. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |

| | |
|---|---|
| #GP(selector) | If a code segment or gate or TSS selector index is outside descriptor table limits. |
| | If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment. |
| | If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL. |
| | If the DPL for a conforming-code segment is greater than the CPL. |
| | If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector. |
| | If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment. |
| | If the segment selector from a call gate is beyond the descriptor table limits. |
| | If the DPL for a code-segment obtained from a call gate is greater than the CPL. |
| | If the segment selector for a TSS has its local/global bit set for local. |
| | If a TSS segment descriptor specifies that the TSS is busy or not available. |
| #SS(0) | If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs. |
| | If a memory operand effective address is outside the SS segment limit. |
| #SS(selector) | If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs. |
| | If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present. |
| | If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs. |
| #NP(selector) | If a code segment, data segment, stack segment, call gate, task gate, or TSS is not present. |
| #TS(selector) | If the new stack segment selector and ESP are beyond the end of the TSS. |
| | If the new stack segment selector is NULL. |

| | |
|---|---|
| | If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed. |
| | If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor. |
| | If the new stack segment is not a writable data segment. |
| | If segment-selector index for stack segment is outside descriptor table limits. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the target offset is beyond the code segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the target offset is beyond the code segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

| | |
|---|---|
| #GP(selector) | If a memory address accessed by the selector is in non-canonical space. |
| #GP(0) | If the target offset in the destination operand is non-canonical. |

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory address is non-canonical. |
| | If target offset in destination operand is non-canonical. |
| | If the segment selector in the destination operand is NULL. |
| | If the code segment selector in the 64-bit gate is NULL. |
| #GP(selector) | If code segment or 64-bit call gate is outside descriptor table limits. |

| | |
|---|---|
| | If code segment or 64-bit call gate overlaps non-canonical space. |
| | If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate. |
| | If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L- bit set. |
| | If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL. |
| | If the DPL for a conforming-code segment is greater than the CPL. |
| | If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate. |
| | If the upper type field of a 64-bit call gate is not 0x0. |
| | If the segment selector from a 64-bit call gate is beyond the descriptor table limits. |
| | If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL. |
| | If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear. |
| | If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment. |
| #SS(0) | If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs. |
| | If a memory operand effective address is outside the SS segment limit. |
| | If the stack address is in a non-canonical form. |
| #SS(selector) | If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs. |
| #NP(selector) | If a code segment or 64-bit call gate is not present. |
| #TS(selector) | If the load of the new RSP exceeds the limit of the TSS. |
| #UD | (64-bit mode only) If a far call is direct to an absolute address in memory. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 98 | CBW | Valid | Valid | AX ←sign-extend of AL. |
| 98 | CWDE | Valid | Valid | EAX ←sign-extend of AX. |
| REX.W + 98 | CDQE | Valid | N.E. | RAX ←sign-extend of EAX. |

## Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the high 16 bits of the EAX register.

CBW and CWDE reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16; CWDE is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size. Others may treat these two mnemonics as synonyms (CBW/CWDE) and use the setting of the operand-size attribute to determine the size of values to be converted.

In 64-bit mode, the default operation size is the size of the destination register. Use of the REX.W prefix promotes this instruction (CDQE when promoted) to operate on 64-bit operands. In which case, CDQE copies the sign (bit 31) of the doubleword in the EAX register into the high 32 bits of RAX.

## Operation

```
IF OperandSize = 16 (* Instruction = CBW *)
    THEN
        AX ←SignExtend(AL);
    ELSE IF (OperandSize = 32, Instruction = CWDE)
        EAX ←SignExtend(AX); FI;
    ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)
        RAX ←SignExtend(EAX);
FI;
```

## Flags Affected

None.

## Exceptions (All Operating Modes)

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

# CLC—Clear Carry Flag

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F8 | CLC | Valid | Valid | Clear CF flag. |

## Description

Clears the CF flag in the EFLAGS register. Operation is the same in all non-64-bit modes and 64-bit mode.

## Operation

CF ←0;

## Flags Affected

The CF flag is set to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

## Exceptions (All Operating Modes)

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

# CLD—Clear Direction Flag

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| FC | CLD | Valid | Valid | Clear DF flag. |

## Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI). Operation is the same in all non-64-bit modes and 64-bit mode.

## Operation

DF ←0;

## Flags Affected

The DF flag is set to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

## Exceptions (All Operating Modes)

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

# CLFLUSH—Flush Cache Line

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F AE /7 | CLFLUSH *m8* | Valid | Valid | Flushes cache line containing *m8*. |

## Description

Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see "CPUID—CPU Identification" in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH*h* instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH*h* instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

CLFLUSH is only ordered by the MFENCE instruction. It is not guaranteed to be ordered by any other fencing or serializing instructions or by another CLFLUSH instruction. For example, software can use an MFENCE instruction to insure that previous stores are included in the write-back.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
Flush_Cache_Line(SRC);
```

## Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSH  void _mm_clflush(void const *p)

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #UD | If CPUID.01H:EDX.CLFSH[bit 19] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #UD | If CPUID.01H:EDX.CLFSH[bit 19] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #UD | If CPUID.01H:EDX.CLFSH[bit 19] = 0. |
| | If the LOCK prefix is used. |

# CLI — Clear Interrupt Flag

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| FA | CLI | Valid | Valid | Clear interrupt flag; interrupts disabled when interrupt flag cleared. |

## Description

If protected-mode virtual interrupts are not enabled, CLI clears the IF flag in the EFLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no affect on the generation of exceptions and NMI interrupts.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected. Table 3-6 indicates the action of the CLI instruction depending on the processor operating mode and the CPL/IOPL of the running program or procedure.

CLI operation is the same in non-64-bit modes and 64-bit mode.

**Table 3-6. Decision Table for CLI Results**

| PE | VM | IOPL | CPL | PVI | VIP | VME | CLI Result |
|---|---|---|---|---|---|---|---|
| 0 | X | X | X | X | X | X | **IF = 0** |
| 1 | 0 | ≥ CPL | X | X | X | X | **IF = 0** |
| 1 | 0 | < CPL | 3 | 1 | X | X | **VIF = 0** |
| 1 | 0 | < CPL | < 3 | X | X | X | **GP Fault** |
| 1 | 0 | < CPL | X | 0 | X | X | **GP Fault** |
| 1 | 1 | 3 | X | X | X | X | **IF = 0** |
| 1 | 1 | < 3 | X | X | X | 1 | **VIF = 0** |
| 1 | 1 | < 3 | X | X | X | 0 | **GP Fault** |

**NOTES:**

* X = This setting has no impact.

## Operation

```
IF PE = 0
    THEN
        IF ←0; (* Reset Interrupt Flag *)
    ELSE
        IF VM = 0;
```

```
            THEN
                IF IOPL ←CPL
                    THEN
                        IF ← 0; (* Reset Interrupt Flag *)
                    ELSE
                        IF ((IOPL < CPL) and (CPL = 3) and (PVI = 1))
                            THEN
                                VIF ←0; (* Reset Virtual Interrupt Flag *)
                            ELSE
                                #GP(0);
                        FI;
                FI;
            ELSE (* VM = 1 *)
                IF IOPL = 3
                    THEN
                        IF ←0; (* Reset Interrupt Flag *)
                    ELSE
                        IF (IOPL < 3) AND (VME = 1)
                            THEN
                                VIF ←0; (* Reset Virtual Interrupt Flag *)
                            ELSE
                                #GP(0);
                        FI;
                FI;
        FI;
FI;
```

## Flags Affected

If protected-mode virtual interrupts are not enabled, IF is set to 0 if the CPL is equal to or less than the IOPL; otherwise, it is not affected. The other flags in the EFLAGS register are unaffected.

When protected-mode virtual interrupts are enabled, CPL is 3, and IOPL is less than 3; CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the CPL is greater (has less privilege) than the IOPL of the current program or procedure. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If the CPL is greater (has less privilege) than the IOPL of the current program or procedure. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the CPL is greater (has less privilege) than the IOPL of the current program or procedure. |
| #UD | If the LOCK prefix is used. |

# CLTS—Clear Task-Switched Flag in CR0

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 06 | CLTS | Valid | Valid | Clears TS flag in CR0. |

## Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled "Control Registers" in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information about this flag.

CLTS operation is the same in non-64-bit modes and 64-bit mode.

See Chapter 21, "VMX Non-Root Operation," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

CR0.TS[bit 3] ←0;

## Flags Affected

The TS flag in CR0 register is cleared.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | CLTS is not recognized in virtual-8086 mode. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the CPL is greater than 0. |
| #UD | If the LOCK prefix is used. |

# CMC—Complement Carry Flag

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F5 | CMC | Valid | Valid | Complement CF flag. |

## Description

Complements the CF flag in the EFLAGS register. CMC operation is the same in non-64-bit modes and 64-bit mode.

## Operation

EFLAGS.CF[bit 0]←NOT EFLAGS.CF[bit 0];

## Flags Affected

The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

## Exceptions (All Operating Modes)

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## CMOV*cc*—Conditional Move

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 47 */r* | CMOVA *r16, r/m16* | Valid | Valid | Move if above (CF=0 and ZF=0). |
| 0F 47 */r* | CMOVA *r32, r/m32* | Valid | Valid | Move if above (CF=0 and ZF=0). |
| REX.W + 0F 47 */r* | CMOVA *r64, r/m64* | Valid | N.E. | Move if above (CF=0 and ZF=0). |
| 0F 43 */r* | CMOVAE *r16, r/m16* | Valid | Valid | Move if above or equal (CF=0). |
| 0F 43 */r* | CMOVAE *r32, r/m32* | Valid | Valid | Move if above or equal (CF=0). |
| REX.W + 0F 43 */r* | CMOVAE *r64, r/m64* | Valid | N.E. | Move if above or equal (CF=0). |
| 0F 42 */r* | CMOVB *r16, r/m16* | Valid | Valid | Move if below (CF=1). |
| 0F 42 */r* | CMOVB *r32, r/m32* | Valid | Valid | Move if below (CF=1). |
| REX.W + 0F 42 */r* | CMOVB *r64, r/m64* | Valid | N.E. | Move if below (CF=1). |
| 0F 46 */r* | CMOVBE *r16, r/m16* | Valid | Valid | Move if below or equal (CF=1 or ZF=1). |
| 0F 46 */r* | CMOVBE *r32, r/m32* | Valid | Valid | Move if below or equal (CF=1 or ZF=1). |
| REX.W + 0F 46 */r* | CMOVBE *r64, r/m64* | Valid | N.E. | Move if below or equal (CF=1 or ZF=1). |
| 0F 42 */r* | CMOVC *r16, r/m16* | Valid | Valid | Move if carry (CF=1). |
| 0F 42 */r* | CMOVC *r32, r/m32* | Valid | Valid | Move if carry (CF=1). |
| REX.W + 0F 42 */r* | CMOVC *r64, r/m64* | Valid | N.E. | Move if carry (CF=1). |
| 0F 44 */r* | CMOVE *r16, r/m16* | Valid | Valid | Move if equal (ZF=1). |
| 0F 44 */r* | CMOVE *r32, r/m32* | Valid | Valid | Move if equal (ZF=1). |
| REX.W + 0F 44 */r* | CMOVE *r64, r/m64* | Valid | N.E. | Move if equal (ZF=1). |
| 0F 4F */r* | CMOVG *r16, r/m16* | Valid | Valid | Move if greater (ZF=0 and SF=OF). |
| 0F 4F */r* | CMOVG *r32, r/m32* | Valid | Valid | Move if greater (ZF=0 and SF=OF). |
| REX.W + 0F 4F */r* | CMOVG *r64, r/m64* | Valid | N.E. | Move if greater (ZF=0 and SF=OF). |
| 0F 4D */r* | CMOVGE *r16, r/m16* | Valid | Valid | Move if greater or equal (SF=OF). |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 4D /r | CMOVGE *r32, r/m32* | Valid | Valid | Move if greater or equal (SF=OF). |
| REX.W + 0F 4D /r | CMOVGE *r64, r/m64* | Valid | N.E. | Move if greater or equal (SF=OF). |
| 0F 4C /r | CMOVL *r16, r/m16* | Valid | Valid | Move if less (SF≠ OF). |
| 0F 4C /r | CMOVL *r32, r/m32* | Valid | Valid | Move if less (SF≠ OF). |
| REX.W + 0F 4C /r | CMOVL *r64, r/m64* | Valid | N.E. | Move if less (SF≠ OF). |
| 0F 4E /r | CMOVLE *r16, r/m16* | Valid | Valid | Move if less or equal (ZF=1 or SF≠ OF). |
| 0F 4E /r | CMOVLE *r32, r/m32* | Valid | Valid | Move if less or equal (ZF=1 or SF≠ OF). |
| REX.W + 0F 4E /r | CMOVLE *r64, r/m64* | Valid | N.E. | Move if less or equal (ZF=1 or SF≠ OF). |
| 0F 46 /r | CMOVNA *r16, r/m16* | Valid | Valid | Move if not above (CF=1 or ZF=1). |
| 0F 46 /r | CMOVNA *r32, r/m32* | Valid | Valid | Move if not above (CF=1 or ZF=1). |
| REX.W + 0F 46 /r | CMOVNA *r64, r/m64* | Valid | N.E. | Move if not above (CF=1 or ZF=1). |
| 0F 42 /r | CMOVNAE *r16, r/m16* | Valid | Valid | Move if not above or equal (CF=1). |
| 0F 42 /r | CMOVNAE *r32, r/m32* | Valid | Valid | Move if not above or equal (CF=1). |
| REX.W + 0F 42 /r | CMOVNAE *r64, r/m64* | Valid | N.E. | Move if not above or equal (CF=1). |
| 0F 43 /r | CMOVNB *r16, r/m16* | Valid | Valid | Move if not below (CF=0). |
| 0F 43 /r | CMOVNB *r32, r/m32* | Valid | Valid | Move if not below (CF=0). |
| REX.W + 0F 43 /r | CMOVNB *r64, r/m64* | Valid | N.E. | Move if not below (CF=0). |
| 0F 47 /r | CMOVNBE *r16, r/m16* | Valid | Valid | Move if not below or equal (CF=0 and ZF=0). |
| 0F 47 /r | CMOVNBE *r32, r/m32* | Valid | Valid | Move if not below or equal (CF=0 and ZF=0). |
| REX.W + 0F 47 /r | CMOVNBE *r64, r/m64* | Valid | N.E. | Move if not below or equal (CF=0 and ZF=0). |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 43 /r | CMOVNC *r16, r/m16* | Valid | Valid | Move if not carry (CF=0). |
| 0F 43 /r | CMOVNC *r32, r/m32* | Valid | Valid | Move if not carry (CF=0). |
| REX.W + 0F 43 /r | CMOVNC *r64, r/m64* | Valid | N.E. | Move if not carry (CF=0). |
| 0F 45 /r | CMOVNE *r16, r/m16* | Valid | Valid | Move if not equal (ZF=0). |
| 0F 45 /r | CMOVNE *r32, r/m32* | Valid | Valid | Move if not equal (ZF=0). |
| REX.W + 0F 45 /r | CMOVNE *r64, r/m64* | Valid | N.E. | Move if not equal (ZF=0). |
| 0F 4E /r | CMOVNG *r16, r/m16* | Valid | Valid | Move if not greater (ZF=1 or SF≠ OF). |
| 0F 4E /r | CMOVNG *r32, r/m32* | Valid | Valid | Move if not greater (ZF=1 or SF≠ OF). |
| REX.W + 0F 4E /r | CMOVNG *r64, r/m64* | Valid | N.E. | Move if not greater (ZF=1 or SF≠ OF). |
| 0F 4C /r | CMOVNGE *r16, r/m16* | Valid | Valid | Move if not greater or equal (SF≠ OF). |
| 0F 4C /r | CMOVNGE *r32, r/m32* | Valid | Valid | Move if not greater or equal (SF≠ OF). |
| REX.W + 0F 4C /r | CMOVNGE *r64, r/m64* | Valid | N.E. | Move if not greater or equal (SF≠ OF). |
| 0F 4D /r | CMOVNL *r16, r/m16* | Valid | Valid | Move if not less (SF=OF). |
| 0F 4D /r | CMOVNL *r32, r/m32* | Valid | Valid | Move if not less (SF=OF). |
| REX.W + 0F 4D /r | CMOVNL *r64, r/m64* | Valid | N.E. | Move if not less (SF=OF). |
| 0F 4F /r | CMOVNLE *r16, r/m16* | Valid | Valid | Move if not less or equal (ZF=0 and SF=OF). |
| 0F 4F /r | CMOVNLE *r32, r/m32* | Valid | Valid | Move if not less or equal (ZF=0 and SF=OF). |
| REX.W + 0F 4F /r | CMOVNLE *r64, r/m64* | Valid | N.E. | Move if not less or equal (ZF=0 and SF=OF). |
| 0F 41 /r | CMOVNO *r16, r/m16* | Valid | Valid | Move if not overflow (OF=0). |
| 0F 41 /r | CMOVNO *r32, r/m32* | Valid | Valid | Move if not overflow (OF=0). |
| REX.W + 0F 41 /r | CMOVNO *r64, r/m64* | Valid | N.E. | Move if not overflow (OF=0). |
| 0F 4B /r | CMOVNP *r16, r/m16* | Valid | Valid | Move if not parity (PF=0). |
| 0F 4B /r | CMOVNP *r32, r/m32* | Valid | Valid | Move if not parity (PF=0). |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| REX.W + 0F 4B /r | CMOVNP *r64, r/m64* | Valid | N.E. | Move if not parity (PF=0). |
| 0F 49 /r | CMOVNS *r16, r/m16* | Valid | Valid | Move if not sign (SF=0). |
| 0F 49 /r | CMOVNS *r32, r/m32* | Valid | Valid | Move if not sign (SF=0). |
| REX.W + 0F 49 /r | CMOVNS *r64, r/m64* | Valid | N.E. | Move if not sign (SF=0). |
| 0F 45 /r | CMOVNZ *r16, r/m16* | Valid | Valid | Move if not zero (ZF=0). |
| 0F 45 /r | CMOVNZ *r32, r/m32* | Valid | Valid | Move if not zero (ZF=0). |
| REX.W + 0F 45 /r | CMOVNZ *r64, r/m64* | Valid | N.E. | Move if not zero (ZF=0). |
| 0F 40 /r | CMOVO *r16, r/m16* | Valid | Valid | Move if overflow (OF=0). |
| 0F 40 /r | CMOVO *r32, r/m32* | Valid | Valid | Move if overflow (OF=0). |
| REX.W + 0F 40 /r | CMOVO *r64, r/m64* | Valid | N.E. | Move if overflow (OF=0). |
| 0F 4A /r | CMOVP *r16, r/m16* | Valid | Valid | Move if parity (PF=1). |
| 0F 4A /r | CMOVP *r32, r/m32* | Valid | Valid | Move if parity (PF=1). |
| REX.W + 0F 4A /r | CMOVP *r64, r/m64* | Valid | N.E. | Move if parity (PF=1). |
| 0F 4A /r | CMOVPE *r16, r/m16* | Valid | Valid | Move if parity even (PF=1). |
| 0F 4A /r | CMOVPE *r32, r/m32* | Valid | Valid | Move if parity even (PF=1). |
| REX.W + 0F 4A /r | CMOVPE *r64, r/m64* | Valid | N.E. | Move if parity even (PF=1). |
| 0F 4B /r | CMOVPO *r16, r/m16* | Valid | Valid | Move if parity odd (PF=0). |
| 0F 4B /r | CMOVPO *r32, r/m32* | Valid | Valid | Move if parity odd (PF=0). |
| REX.W + 0F 4B /r | CMOVPO *r64, r/m64* | Valid | N.E. | Move if parity odd (PF=0). |
| 0F 48 /r | CMOVS *r16, r/m16* | Valid | Valid | Move if sign (SF=1). |
| 0F 48 /r | CMOVS *r32, r/m32* | Valid | Valid | Move if sign (SF=1). |
| REX.W + 0F 48 /r | CMOVS *r64, r/m64* | Valid | N.E. | Move if sign (SF=1). |
| 0F 44 /r | CMOVZ *r16, r/m16* | Valid | Valid | Move if zero (ZF=1). |
| 0F 44 /r | CMOVZ *r32, r/m32* | Valid | Valid | Move if zero (ZF=1). |
| REX.W + 0F 44 /r | CMOVZ *r64, r/m64* | Valid | N.E. | Move if zero (ZF=1). |

## Description

The CMOV*cc* instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOV*cc* instruction.

These instructions can move 16-bit, 32-bit or 64-bit values from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The condition for each CMOV*cc* mnemonic is given in the description column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.

The CMOV*cc* instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOV*cc* instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in this chapter).

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
temp ←SRC

IF (64-Bit Mode)
    THEN
        IF condition TRUE
            THEN
                IF (OperandSize = 64)
                    THEN
                        DEST ←temp;
                    ELSE
                        DEST ←temp AND 0x00000000_FFFFFFFF;
                FI;
        FI;
ELSE
    IF condition TRUE
        THEN
```

```
            DEST ←temp;
    FI;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## CMP—Compare Two Operands

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 3C *ib* | CMP AL, *imm8* | Valid | Valid | Compare *imm8* with AL. |
| 3D *iw* | CMP AX, *imm16* | Valid | Valid | Compare *imm16* with AX. |
| 3D *id* | CMP EAX, *imm32* | Valid | Valid | Compare *imm32* with EAX. |
| REX.W + 3D *id* | CMP RAX, *imm32* | Valid | N.E. | Compare *imm32 sign-extended to 64-bits* with RAX. |
| 80 /7 *ib* | CMP *r/m8, imm8* | Valid | Valid | Compare *imm8* with *r/m8.* |
| REX + 80 /7 *ib* | CMP *r/m8*, imm8* | Valid | N.E. | Compare *imm8* with *r/m8.* |
| 81 /7 *iw* | CMP *r/m16, imm16* | Valid | Valid | Compare *imm16* with *r/m16.* |
| 81 /7 *id* | CMP *r/m32, imm32* | Valid | Valid | Compare *imm32* with *r/m32.* |
| REX.W + 81 /7 *id* | CMP *r/m64, imm32* | Valid | N.E. | Compare *imm32 sign-extended to 64-bits* with *r/m64.* |
| 83 /7 *ib* | CMP *r/m16, imm8* | Valid | Valid | Compare *imm8* with *r/m16.* |
| 83 /7 *ib* | CMP *r/m32, imm8* | Valid | Valid | Compare *imm8* with *r/m32.* |
| REX.W + 83 /7 *ib* | CMP *r/m64, imm8* | Valid | N.E. | Compare *imm8* with *r/m64.* |
| 38 /*r* | CMP *r/m8, r8* | Valid | Valid | Compare *r8* with *r/m8.* |
| REX + 38 /*r* | CMP *r/m8*, r8** | Valid | N.E. | Compare *r8* with *r/m8.* |
| 39 /*r* | CMP *r/m16, r16* | Valid | Valid | Compare *r16* with *r/m16.* |
| 39 /*r* | CMP *r/m32, r32* | Valid | Valid | Compare *r32* with *r/m32.* |
| REX.W + 39 /*r* | CMP *r/m64,r64* | Valid | N.E. | Compare *r64* with *r/m64.* |
| 3A /*r* | CMP *r8, r/m8* | Valid | Valid | Compare *r/m8* with *r8.* |
| REX + 3A /*r* | CMP *r8*, r/m8** | Valid | N.E. | Compare *r/m8 with r8.* |
| 3B /*r* | CMP *r16, r/m16* | Valid | Valid | Compare *r/m16* with *r16.* |
| 3B /*r* | CMP *r32, r/m32* | Valid | Valid | Compare *r/m32* with *r32.* |
| REX.W + 3B /*r* | CMP *r64, r/m64* | Valid | N.E. | Compare *r/m64* with *r64.* |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction. Appendix B, "EFLAGS Condition Codes," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, shows the relationship of the status flags and the condition codes.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
temp ←SRC1 –SignExtend(SRC2);
ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)
```

## Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# CMPPD—Compare Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F C2 /*r* ib | CMPPD *xmm1, xmm2/m128, imm8* | Valid | Valid | Compare packed double-precision floating-point values in *xmm2/m128* and *xmm1* using imm8 as comparison predicate. |

## Description

Performs a SIMD compare of the two packed double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The comparison predicate operand is an 8-bit immediate, the first 3 bits of which define the type of comparison to be made (see Table 3-7). Bits 4 through 7 of the immediate are reserved.

**Table 3-7. Comparison Predicate for CMPPD and CMPPS Instructions**

| Predicate | imm8 Encoding | Description | Relation where: A Is 1st Operand B Is 2nd Operand | Emulation | Result if NaN Operand | QNaN Operand Signals Invalid |
|---|---|---|---|---|---|---|
| EQ | 000B | Equal | A = B | | False | No |
| LT | 001B | Less-than | A < B | | False | Yes |
| LE | 010B | Less-than-or-equal | A ≤B | | False | Yes |
| | | Greater than | A >B | Swap Operands, Use LT | False | Yes |
| | | Greater-than-or-equal | A ≥ B | Swap Operands, Use LE | False | Yes |
| UNORD | 011B | Unordered | A, B = Unordered | | True | No |
| NEQ | 100B | Not-equal | A ≠ B | | True | No |
| NLT | 101B | Not-less-than | NOT(A < B) | | True | Yes |

**Table 3-7. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)**

| Predi-cate | imm8 Encod-ing | Description | Relation where: A Is 1st Operand B Is 2nd Operand | Emulation | Result if NaN Operand | QNaN Oper-and Signals Invalid |
|---|---|---|---|---|---|---|
| NLE | 110B | Not-less-than-or-equal | NOT(A ≤B) | | True | Yes |
| | | Not-greater-than | NOT(A > B) | Swap Operands, Use NLT | True | Yes |
| | | Not-greater-than-or-equal | NOT(A ≥ B) | Swap Operands, Use NLE | True | Yes |
| ORD | 111B | Ordered | A , B = Ordered | | False | No |

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that the processor does not implement the greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction. See Table 3-7.

**Table 3-8. Pseudo-Op and CMPPD Implementation**

| Pseudo-Op | CMPPD Implementation |
|---|---|
| CMPEQPD *xmm1, xmm2* | CMPPD *xmm1, xmm2, 0* |
| CMPLTPD *xmm1, xmm2* | CMPPD *xmm1, xmm2, 1* |
| CMPLEPD *xmm1, xmm2* | CMPPD *xmm1, xmm2, 2* |
| CMPUNORDPD *xmm1, xmm2* | CMPPD *xmm1, xmm2, 3* |
| CMPNEQPD *xmm1, xmm2* | CMPPD *xmm1, xmm2, 4* |
| CMPNLTPD *xmm1, xmm2* | CMPPD *xmm1, xmm2, 5* |

**Table 3-8. Pseudo-Op and CMPPD Implementation**

| Pseudo-Op | CMPPD Implementation |
|---|---|
| CMPNLEPD *xmm1, xmm2* | CMPPD *xmm1, xmm2, 6* |
| CMPORDPD *xmm1, xmm2* | CMPPD *xmm1, xmm2, 7* |

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
CASE (COMPARISON PREDICATE) OF
    0:    OP ←EQ;
    1:    OP ←LT;
    2:    OP ←LE;
    3:    OP ←UNORD;
    4:    OP ←NEQ;
    5:    OP ←NLT;
    6:    OP ←NLE;
    7:    OP ←ORD;
    DEFAULT: Reserved;

CMP0 ←DEST[63:0]  OP  SRC[63:0];
CMP1 ←DEST[127:64]  OP  SRC[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ←0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ←0000000000000000H; FI;
```

## Intel C/C++ Compiler Intrinsic Equivalents

CMPPD for equality __m128d _mm_cmpeq_pd(__m128d a, __m128d b)

CMPPD for less-than __m128d _mm_cmplt_pd(__m128d a, __m128d b)

CMPPD for less-than-or-equal __m128d _mm_cmple_pd(__m128d a, __m128d b)

CMPPD for greater-than __m128d _mm_cmpgt_pd(__m128d a, __m128d b)

CMPPD for greater-than-or-equal __m128d _mm_cmpge_pd(__m128d a, __m128d b)

| | |
|---|---|
| CMPPD for inequality | __m128d _mm_cmpneq_pd(__m128d a, __m128d b) |
| CMPPD for not-less-than | __m128d _mm_cmpnlt_pd(__m128d a, __m128d b) |
| CMPPD for not-greater-than | __m128d _mm_cmpngt_pd(__m128d a, __m128d b) |
| CMPPD for not-greater-than-or-equal | __m128d _mm_cmpnge_pd(__m128d a, __m128d b) |
| CMPPD for ordered | __m128d _mm_cmpord_pd(__m128d a, __m128d b) |
| CMPPD for unordered | __m128d _mm_cmpunord_pd(__m128d a, __m128d b) |
| CMPPD for not-less-than-or-equal | __m128d _mm_cmpnle_pd(__m128d a, __m128d b) |

## SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in above table, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment.<br>If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form.<br>If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# CMPPS—Compare Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F C2 /*r* ib | CMPPS *xmm1, xmm2/m128, imm8* | Valid | Valid | Compare packed single-precision floating-point values in *xmm2/mem* and *xmm1* using *imm8* as comparison predicate. |

## Description

Performs a SIMD compare of the four packed single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The comparison predicate operand is an 8-bit immediate, the first 3 bits of which define the type of comparison to be made (see Table 3-7). Bits 4 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Some of the comparisons listed in Table 3-7 (such as the greater-than, greater-than-or-equal, not-greater-than, and not-greater-than-or-equal relations) can be made only through software emulation. For these comparisons the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction. See Table 3-9.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

**Table 3-9. Pseudo-Ops and CMPPS**

| Pseudo-Op | Implementation |
|---|---|
| CMPEQPS *xmm1, xmm2* | CMPPS *xmm1, xmm2, 0* |
| CMPLTPS *xmm1, xmm2* | CMPPS *xmm1, xmm2, 1* |
| CMPLEPS *xmm1, xmm2* | CMPPS *xmm1, xmm2, 2* |
| CMPUNORDPS *xmm1, xmm2* | CMPPS *xmm1, xmm2, 3* |
| CMPNEQPS *xmm1, xmm2* | CMPPS *xmm1, xmm2, 4* |
| CMPNLTPS *xmm1, xmm2* | CMPPS *xmm1, xmm2, 5* |
| CMPNLEPS *xmm1, xmm2* | CMPPS *xmm1, xmm2, 6* |
| CMPORDPS *xmm1, xmm2* | CMPPS *xmm1, xmm2, 7* |

The greater-than relations not implemented by the processor require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

## Operation

```
CASE (COMPARISON PREDICATE) OF
    0:   OP ←EQ;
    1:   OP ←LT;
    2:   OP ←LE;
    3:   OP ←UNORD;
    4:   OP ←NE;
    5:   OP ←NLT;
    6:   OP ←NLE;
    7:   OP ←ORD;
EASC;

CMP0 ←DEST[31:0]  OP  SRC[31:0];
CMP1 ←DEST[63:32]  OP  SRC[63:32];
CMP2 ←DEST [95:64]  OP  SRC[95:64];
CMP3 ←DEST[127:96]  OP  SRC[127:96];

IF CMP0 = TRUE
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ←00000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ←FFFFFFFFH;
    ELSE DEST[63:32] ←00000000H; FI;
IF CMP2 = TRUE
```

```
    THEN DEST95:64] ←FFFFFFFFH;
    ELSE DEST[95:64] ←00000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ←FFFFFFFFH;
    ELSE DEST[127:96] ←00000000H; FI;
```

## Intel C/C++ Compiler Intrinsic Equivalents

| | |
|---|---|
| CMPPS for equality | __m128 _mm_cmpeq_ps(__m128 a, __m128 b) |
| CMPPS for less-than | __m128 _mm_cmplt_ps(__m128 a, __m128 b) |
| CMPPS for less-than-or-equal | __m128 _mm_cmple_ps(__m128 a, __m128 b) |
| CMPPS for greater-than | __m128 _mm_cmpgt_ps(__m128 a, __m128 b) |
| CMPPS for greater-than-or-equal | __m128 _mm_cmpge_ps(__m128 a, __m128 b) |
| CMPPS for inequality | __m128 _mm_cmpneq_ps(__m128 a, __m128 b) |
| CMPPS for not-less-than | __m128 _mm_cmpnlt_ps(__m128 a, __m128 b) |
| CMPPS for not-greater-than | __m128 _mm_cmpngt_ps(__m128 a, __m128 b) |
| CMPPS for not-greater-than-or-equal | __m128 _mm_cmpnge_ps(__m128 a, __m128 b) |
| CMPPS for ordered | __m128 _mm_cmpord_ps(__m128 a, __m128 b) |
| CMPPS for unordered | __m128 _mm_cmpunord_ps(__m128 a, __m128 b) |
| CMPPS for not-less-than-or-equal | __m128 _mm_cmpnle_ps(__m128 a, __m128 b) |

## SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in above table, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |

| | |
|---|---|
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

## CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| A6 | CMPS *m8, m8* | Valid | Valid | For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R\|E)SI to byte at address (R\|E)DI. The status flags are set accordingly. |
| A7 | CMPS *m16, m16* | Valid | Valid | For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R\|E)SI with word at address (R\|E)DI. The status flags are set accordingly. |
| A7 | CMPS *m32, m32* | Valid | Valid | For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R\|E)SI at dword at address (R\|E)DI. The status flags are set accordingly. |
| REX.W + A7 | CMPS *m64, m64* | Valid | N.E. | Compares quadword at address (R\|E)SI with quadword at address (R\|E)DI and sets the status flags accordingly. |
| A6 | CMPSB | Valid | Valid | For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R\|E)SI with byte at address (R\|E)DI. The status flags are set accordingly. |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| A7 | CMPSW | Valid | Valid | For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R\|E)SI with word at address (R\|E)DI. The status flags are set accordingly. |
| A7 | CMPSD | Valid | Valid | For legacy mode, compare dword at address DS:(E)SI with dword at address ES:(E)DI; For 64-bit mode compare dword at address (R\|E)SI with dword at address (R\|E)DI. The status flags are set accordingly. |
| REX.W + A7 | CMPSQ | Valid | N.E. | Compares quadword at address (R\|E)SI with quadword at address (R\|E)DI and sets the status flags accordingly. |

## Description

Compares the byte, word, doubleword, or quadword specified with the first source operand with the byte, word, doubleword, or quadword specified with the second source operand and sets the status flags in the EFLAGS register according to the results.

Both source operands are located in memory. The address of the first source operand is read from DS:SI, DS:ESI or RSI (depending on the address-size attribute of the instruction is 16, 32, or 64, respectively). The address of the second source operand is read from ES:DI, ES:EDI or RDI (again depending on the address-size attribute of the instruction is 16, 32, or 64). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operand form is provided to allow documentation. However, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords, quadwords), but they do not have to specify the correct location. Locations of the source operands are always specified by the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI (or RSI) and ES:(E)DI (or

RDI) registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), CMPSD (doubleword comparison), or CMPSQ (quadword comparison using REX.W).

After the comparison, the (E/R)SI and (E/R)DI registers increment or decrement automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E/R)SI and (E/R)DI register increment; if the DF flag is 1, the registers decrement.) The registers increment or decrement by 1 for byte operations, by 2 for word operations, 4 for doubleword operations. If operand size is 64, RSI and RDI registers increment by 8 for quadword operations.

The CMPS, CMPSB, CMPSW, CMPSD, and CMPSQ instructions can be preceded by the REP prefix for block comparisons. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See "REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix.

In 64-bit mode, the instruction's default address size is 64 bits, 32 bit address size is supported using the prefix 67H. Use of the REX.W prefix promotes doubleword operation to 64 bits (see CMPSQ). See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
temp SRC1 - SRC2;
SetStatusFlags(temp);

IF (64-Bit Mode)
    THEN
        IF (Byte comparison)
        THEN IF DF = 0
            THEN
                (R|E)SI ←(R|E)SI +1;
                (R|E)DI ←(R|E)DI +1;
            ELSE
                (R|E)SI ←(R|E)SI – 1;
                (R|E)DI ←(R|E)DI – 1;
            FI;
        ELSE IF (Word comparison)
            THEN IF DF = 0
                THEN
                    (R|E)SI ←(R|E)SI +2;
                    (R|E)DI ←(R|E)DI +2;
                ELSE
                    (R|E)SI ←(R|E)SI – 2;
```

```
                (R|E)DI ←(R|E)DI – 2;
            FI;
        ELSE IF (Doubleword comparison)
            THEN IF DF = 0
                THEN
                    (R|E)SI ←(R|E)SI +4;
                    (R|E)DI ←(R|E)DI +4;
                ELSE
                    (R|E)SI ←(R|E)SI – 4;
                    (R|E)DI ←(R|E)DI – 4;
                FI;
        ELSE (* Quadword comparison *)
            THEN IF DF = 0
                (R|E)SI ←(R|E)SI +8;
                (R|E)DI ←(R|E)DI +8;
            ELSE
                (R|E)SI ←(R|E)SI – 8;
                (R|E)DI ←(R|E)DI – 8;
            FI;
        FI;
    ELSE (* Non-64-bit Mode *)
        IF (byte comparison)
        THEN IF DF = 0
            THEN
                (E)SI ←(E)SI +1;
                (E)DI ←(E)DI +1;
            ELSE
                (E)SI ←(E)SI – 1;
                (E)DI ←(E)DI – 1;
            FI;
        ELSE IF (Word comparison)
            THEN IF DF = 0
                (E)SI ←(E)SI +2;
                (E)DI ←(E)DI +2;
            ELSE
                (E)SI ←(E)SI – 2;
                (E)DI ←(E)DI – 2;
            FI;
        ELSE (* Doubleword comparison *)
            THEN IF DF = 0
                (E)SI ←(E)SI +4;
                (E)DI ←(E)DI +4;
            ELSE
```

```
                (E)SI ←(E)SI – 4;
                (E)DI ←(E)DI – 4;
            FI;
        FI;
FI;
```

## Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the temporary result of the comparison.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# CMPSD—Compare Scalar Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F C2 /r ib | CMPSD *xmm1, xmm2/m64, imm8* | Valid | Valid | Compare low double-precision floating-point value in *xmm2/m64* and *xmm1* using *imm8* as comparison predicate. |

## Description

Compares the low double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand; the high quadword remains unchanged. The comparison predicate operand is an 8-bit immediate, the first 3 bits of which define the type of comparison to be made (see Table 3-7). Bits 4 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Some of the comparisons listed in Table 3-7 can be achieved only through software emulation. For these comparisons the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction. See Table 3-10.

**Table 3-10. Pseudo-Ops and CMPSD**

| Pseudo-Op | Implementation |
|---|---|
| CMPEQSD *xmm1, xmm2* | CMPSD *xmm1,xmm2, 0* |
| CMPLTSD *xmm1, xmm2* | CMPSD *xmm1,xmm2, 1* |
| CMPLESD *xmm1, xmm2* | CMPSD *xmm1,xmm2, 2* |

**Table 3-10. Pseudo-Ops and CMPSD (Contd.)**

| Pseudo-Op | Implementation |
|---|---|
| CMPUNORDSD *xmm1, xmm2* | CMPSD *xmm1,xmm2, 3* |
| CMPNEQSD *xmm1, xmm2* | CMPSD *xmm1,xmm2, 4* |
| CMPNLTSD *xmm1, xmm2* | CMPSD *xmm1,xmm2, 5* |
| CMPNLESD *xmm1, xmm2* | CMPSD *xmm1,xmm2, 6* |
| CMPORDSD *xmm1, xmm2* | CMPSD *xmm1,xmm2, 7* |

The greater-than relations not implemented in the processor require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
CASE (COMPARISON PREDICATE) OF
    0:    OP ←EQ;
    1:    OP ←LT;
    2:    OP ←LE;
    3:    OP ←UNORD;
    4:    OP ←NEQ;
    5:    OP ←NLT;
    6:    OP ←NLE;
    7:    OP ←ORD;
    DEFAULT: Reserved;

CMP0 ←DEST[63:0]  OP  SRC[63:0];

IF CMP0 = TRUE
    THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ←0000000000000000H; FI;
(* DEST[127:64] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalents

| | |
|---|---|
| CMPSD for equality | __m128d _mm_cmpeq_sd(__m128d a, __m128d b) |
| CMPSD for less-than | __m128d _mm_cmplt_sd(__m128d a, __m128d b) |
| CMPSD for less-than-or-equal | __m128d _mm_cmple_sd(__m128d a, __m128d b) |
| CMPSD for greater-than | __m128d _mm_cmpgt_sd(__m128d a, __m128d b) |

CMPSD for greater-than-or-equal__m128d _mm_cmpge_sd(__m128d a, __m128d b)
CMPSD for inequality __m128d _mm_cmpneq_sd(__m128d a, __m128d b)
CMPSD for not-less-than __m128d _mm_cmpnlt_sd(__m128d a, __m128d b)
CMPSD for not-greater-than __m128d _mm_cmpngt_sd(__m128d a, __m128d b)
CMPSD for not-greater-than-or-equal__m128d _mm_cmpnge_sd(__m128d a, __m128d b)
CMPSD for ordered __m128d _mm_cmpord_sd(__m128d a, __m128d b)
CMPSD for unordered __m128d _mm_cmpunord_sd(__m128d a, __m128d b)
CMPSD for not-less-than-or-equal__m128d _mm_cmpnle_sd(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in above table, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| #UD | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |

| | |
|---|---|
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXM-MEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXM-MEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CMPSS—Compare Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F C2 */r ib* | CMPSS *xmm1, xmm2/m32, imm8* | Valid | Valid | Compare low single-precision floating-point value in *xmm2/m32* and *xmm1* using *imm8* as comparison predicate. |

## Description

Compares the low single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand; the 3 high-order doublewords remain unchanged. The comparison predicate operand is an 8-bit immediate, the first 3 bits of which define the type of comparison to be made (see Table 3-7). Bits 4 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate a fault, since a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Some of the comparisons listed in Table 3-7 can be achieved only through software emulation. For these comparisons the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction. See Table 3-11.

**Table 3-11. Pseudo-Ops and CMPSS**

| Pseudo-Op | CMPSS Implementation |
|---|---|
| CMPEQSS *xmm1, xmm2* | CMPSS *xmm1, xmm2, 0* |
| CMPLTSS *xmm1, xmm2* | CMPSS *xmm1, xmm2, 1* |
| CMPLESS *xmm1, xmm2* | CMPSS *xmm1, xmm2, 2* |
| CMPUNORDSS *xmm1, xmm2* | CMPSS *xmm1, xmm2, 3* |
| CMPNEQSS *xmm1, xmm2* | CMPSS *xmm1, xmm2, 4* |
| CMPNLTSS *xmm1, xmm2* | CMPSS *xmm1, xmm2, 5* |
| CMPNLESS *xmm1, xmm2* | CMPSS *xmm1, xmm2, 6* |
| CMPORDSS *xmm1, xmm2* | CMPSS *xmm1, xmm2, 7* |

The greater-than relations not implemented in the processor require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
CASE (COMPARISON PREDICATE) OF
    0:    OP ←EQ;
    1:    OP ←LT;
    2:    OP ←LE;
    3:    OP ←UNORD;
    4:    OP ←NEQ;
    5:    OP ←NLT;
    6:    OP ←NLE;
    7:    OP ←ORD;
    DEFAULT: Reserved;

CMP0 ←DEST[31:0]  OP  SRC[31:0];

IF CMP0 = TRUE
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ←00000000H; FI;
(* DEST[127:32] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalents

CMPSS for equality __m128 _mm_cmpeq_ss(__m128 a, __m128 b)

| | |
|---|---|
| CMPSS for less-than | __m128 _mm_cmplt_ss(__m128 a, __m128 b) |
| CMPSS for less-than-or-equal | __m128 _mm_cmple_ss(__m128 a, __m128 b) |
| CMPSS for greater-than | __m128 _mm_cmpgt_ss(__m128 a, __m128 b) |
| CMPSS for greater-than-or-equal | __m128 _mm_cmpge_ss(__m128 a, __m128 b) |
| CMPSS for inequality | __m128 _mm_cmpneq_ss(__m128 a, __m128 b) |
| CMPSS for not-less-than | __m128 _mm_cmpnlt_ss(__m128 a, __m128 b) |
| CMPSS for not-greater-than | __m128 _mm_cmpngt_ss(__m128 a, __m128 b) |
| CMPSS for not-greater-than-or-equal | __m128 _mm_cmpnge_ss(__m128 a, __m128 b) |
| CMPSS for ordered | __m128 _mm_cmpord_ss(__m128 a, __m128 b) |
| CMPSS for unordered | __m128 _mm_cmpunord_ss(__m128 a, __m128 b) |
| CMPSS for not-less-than-or-equal | __m128 _mm_cmpnle_ss(__m128 a, __m128 b) |

## SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in above table, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| #UD | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## CMPXCHG—Compare and Exchange

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F B0/*r* | CMPXCHG *r/m8, r8* | Valid | Valid* | Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL. |
| REX + 0F B0/*r* | CMPXCHG *r/m8**,*r8* | Valid | N.E. | Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL. |
| 0F B1/*r* | CMPXCHG *r/m16, r16* | Valid | Valid* | Compare AX with *r/m16*. If equal, ZF is set and *r16* is loaded into *r/m16*. Else, clear ZF and load *r/m16* into AX. |
| 0F B1/*r* | CMPXCHG *r/m32, r32* | Valid | Valid* | Compare EAX with *r/m32*. If equal, ZF is set and *r32* is loaded into *r/m32*. Else, clear ZF and load *r/m32* into EAX. |
| REX.W + 0F B1/*r* | CMPXCHG *r/m64, r64* | Valid | N.E. | Compare RAX with *r/m64*. If equal, ZF is set and *r64* is loaded into *r/m64*. Else, clear ZF and load *r/m64* into RAX. |

**NOTES:**

* See the IA-32 Architecture Compatibility section below.

** In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Description

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

## Operation

```
(* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word, doubleword, or
quadword comparison is being performed *)

IF accumulator = DEST
    THEN
        ZF ←1;
        DEST ←SRC;
    ELSE
        ZF ←0;
        accumulator ←DEST;
FI;
```

## Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

# CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

| Opcode* | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F C7 /1 *m64* | CMPXCHG8B *m64* | Valid | Valid* | Compare EDX:EAX with *m64*. If equal, set ZF and load ECX:EBX into *m64*. Else, clear ZF and load *m64* into EDX:EAX. |
| REX.W + 0F C7 /1 *m128* | CMPXCHG16B *m128* | Valid | N.E. | Compare RDX:RAX with *m128*. If equal, set ZF and load RCX:RBX into *m128*. Else, clear ZF and load *m128* into RDX:RAX. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Compares the 64-bit value in EDX:EAX (or 128-bit value in RDX:RAX if operand size is 128 bits) with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX (or 128-bit value in RCX:RBX) is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX (or RDX:RAX). The destination operand is an 8-byte memory location (or 16-byte memory location if operand size is 128 bits). For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value. For the RDX:RAX and RCX:RBX register pairs, RDX and RCX contain the high-order 64 bits and RAX and RBX contain the low-order 64bits of a 128-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, default operation size is 64 bits. Use of the REX.W prefix promotes operation to 128 bits. Note that CMPXCHG16B requires that the destination (memory) operand be 16-byte aligned. See the summary chart at the beginning of this section for encoding data and limits. For information on the CPUID flag that indicates CMPXCHG16B, see page 3-173.

## IA-32 Architecture Compatibility

This instruction encoding is not supported on Intel processors earlier than the Pentium processors.

## Operation

```
IF (64-Bit Mode and OperandSize = 64)
    THEN
        IF (RDX:RAX = DEST)
        ZF ←1;
            DEST ←RCX:RBX;
        ELSE
            ZF ←0;
            RDX:RAX ←DEST;
        FI
    ELSE
        IF (EDX:EAX = DEST)
            ZF ←1;
            DEST ←ECX:EBX;
        ELSE
            ZF ←0;
            EDX:EAX ←DEST;
        FI;
FI;
```

## Flags Affected

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

## Protected Mode Exceptions

| | |
|---|---|
| #UD | If the destination is not a memory operand. |
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If the destination operand is not a memory location. |
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| #SS | If a memory operand effective address is outside the SS segment limit. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | If the destination operand is not a memory location. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand for CMPXCHG16B is not aligned on a 16-byte boundary. |
| | IfIf CPUID.01H:ECX.CMPXCHG16B[bit 13] = 0. |
| #UD | If the destination operand is not a memory location. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 2F /r | COMISD *xmm1, xmm2/m64* | Valid | Valid | Compare low double-precision floating-point values in *xmm1* and *xmm2/mem64* and set the EFLAGS flags accordingly. |

## Description

Compares the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
RESULT ←OrderedCompare(DEST[63:0] <>SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED:          ZF,PF,CF ←111;
    GREATER_THAN:       ZF,PF,CF ←000;
    LESS_THAN:          ZF,PF,CF ←001;
    EQUAL:              ZF,PF,CF ←100;
ESAC;
OF, AF, SF ←0; }
```

## Intel C/C++ Compiler Intrinsic Equivalents

int _mm_comieq_sd (__m128d a, __m128d b)

int _mm_comilt_sd (__m128d a, __m128d b)

int _mm_comile_sd (__m128d a, __m128d b)

int _mm_comigt_sd (__m128d a, __m128d b)

int _mm_comige_sd (__m128d a, __m128d b)

int _mm_comineq_sd (__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 2F /r | COMISS *xmm1, xmm2/m32* | Valid | Valid | Compare low single-precision floating-point values in *xmm1* and *xmm2/mem32* and set the EFLAGS flags accordingly. |

## Description

Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF, and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; Operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
RESULT ←OrderedCompare(SRC1[31:0] <> SRC2[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED:          ZF,PF,CF ←111;
    GREATER_THAN:       ZF,PF,CF ←000;
    LESS_THAN:          ZF,PF,CF ←001;
    EQUAL:              ZF,PF,CF ←100;
ESAC;
OF,AF,SF ←0; }
```

## Intel C/C++ Compiler Intrinsic Equivalents

int _mm_comieq_ss (__m128 a, __m128 b)

int _mm_comilt_ss (__m128 a, __m128 b)

int _mm_comile_ss (__m128 a, __m128 b)

int _mm_comigt_ss (__m128 a, __m128 b)

int _mm_comige_ss (__m128 a, __m128 b)

int _mm_comineq_ss (__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CPUID—CPU Identification

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F A2 | CPUID | Valid | Valid | Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well). |

## Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.[1] The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-12 shows information returned, depending on the initial value loaded into the EAX register. Table 3-13 shows the maximum CPUID input value recognized for each family of IA-32 processors on which CPUID is implemented.

Two types of information are returned: basic and extended function information. If a value is entered for CPUID.EAX is invalid for a particular processor, the data for the highest basic information leaf is returned. For example, using the Intel Core 2 Duo processor, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* INVALID: Returns the same information as CPUID.EAX = 0AH. *)
CPUID.EAX = 80000008H (* Returns virtual/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0AH. *)
```

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

**See also:**

"Serializing Instructions" in Chapter 7, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*

*AP-485, Intel Processor Identification and the CPUID Instruction* (Order Number 241618)

**Table 3-12. Information Returned by CPUID Instruction**

| Initial EAX Value | Information Provided about the Processor | |
|---|---|---|
| | *Basic CPUID Information* | |
| 0H | EAX<br>EBX<br>ECX<br>EDX | Maximum Input Value for Basic CPUID Information (see Table 3-13)<br>"Genu"<br>"ntel"<br>"inel" |
| 01H | EAX | Version Information: Type, Family, Model, and Stepping ID (see Figure 3-5) |
| | EBX | Bits 7-0: Brand Index<br>Bits 15-8: CLFLUSH line size (Value ∗ 8 = cache line size in bytes)<br>Bits 23-16: Maximum number of logical processors in this physical package.<br>Bits 31-24: Initial APIC ID |
| | ECX | Extended Feature Information (see Figure 3-6 and Table 3-15) |
| | EDX | Feature Information (see Figure 3-7 and Table 3-16) |
| 02H | EAX<br>EBX<br>ECX<br>EDX | Cache and TLB Information (see Table 3-17)<br>Cache and TLB Information<br>Cache and TLB Information<br>Cache and TLB Information |
| 03H | EAX<br>EBX | Reserved.<br>Reserved. |
| | ECX | Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) |
| | EDX | Bits 32-63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) |
| | | **NOTES:**<br>Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature. |
| | | See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618) for more information on PSN. |

**Table 3-12. Information Returned by CPUID Instruction (Contd.)**

| Initial EAX Value | Information Provided about the Processor |
|---|---|
| | CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default). |
| | *Deterministic Cache Parameters Leaf* |
| 04H | **NOTES:**<br>04H output depends on the initial value in ECX.<br>See also: "INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 3-180.<br><br>EAX Bits 4-0: Cache Type Field<br>0 = Null - No more caches 3 = Unified Cache<br>1 = Data Cache 4-31 = Reserved<br>2 = Instruction Cache<br>Bits 7-5: Cache Level (starts at 1)<br>Bits 8: Self Initializing cache level (does not need SW initialization)<br>Bits 9: Fully Associative cache<br>Bit 10: Write-Back Invalidate/Invalidate<br>0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache<br>1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache.<br>Bit 11: Cache Inclusiveness<br>0 = Cache is not inclusive of lower cache levels.<br>1 = Cache is inclusive of lower cache levels.<br>Bits 13-12: Reserved<br>Bits 25-14: Maximum number of threads sharing this cache in a physical package*<br>Bits 31-26: Maximum number of processor cores in the physical package* **<br>EBX Bits 11-00: L = System Coherency Line Size*<br>Bits 21-12: P = Physical Line partitions*<br>Bits 31-22: W = Ways of associativity*<br>ECX Bits 31-00: S = Number of Sets*<br>EDX Reserved = 0 |
| | **NOTES:**<br>* Add one to the return value to get the result.<br>** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0. |

**Table 3-12. Information Returned by CPUID Instruction (Contd.)**

| Initial EAX Value | Information Provided about the Processor | |
|---|---|---|
| | *MONITOR/MWAIT Leaf* | |
| 5H | EAX | Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity)<br>Bits 31-16: Reserved = 0 |
| | EBX | Bits 15-00: Largest monitor-line size in bytes (default is processor's monitor granularity)<br>Bits 31-16: Reserved = 0 |
| | ECX | Bits 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported<br>Bits 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled<br>Bits 31 - 02: Reserved |
| | EDX | Bits 03 - 00: Number of C0* sub C-states supported using MWait<br>Bits 07 - 04: Number of C1* sub C-states supported using MWAIT<br>Bits 11 - 08: Number of C2* sub C-states supported using MWAIT<br>Bits 15 - 12: Number of C3* sub C-states supported using MWAIT<br>Bits 19 - 16: Number of C4* sub C-states supported using MWAIT<br>Bits 31 - 20: Reserved = 0<br>**NOTE:**<br>* The definition of C0 through C4 states for MWAIT extension are processor-specific C-states, not ACPI C-states. |
| | *Thermal and Power Management Leaf* | |
| 6H | EAX | Bits 00: Digital temperature sensor is supported if set<br>Bits 01: Intel Dynamic Acceleration Enabled<br>Bits 31 - 02: Reserved |
| | EBX | Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor<br>Bits 31 - 04: Reserved |
| | ECX | Bits 00: Hardware Coordination Feedback Capability (Presence of MCNT and ACNT MSRs). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of expected processor performance at frequency specified in CPUID Brand String<br>Bits 31 - 01: Reserved = 0 |
| | EDX | Reserved = 0 |
| | *Architectural Performance Monitoring Leaf* | |

**Table 3-12. Information Returned by CPUID Instruction (Contd.)**

| Initial EAX Value | Information Provided about the Processor | |
|---|---|---|
| 0AH | EAX | Bits 07 - 00: Version ID of architectural performance monitoring<br>Bits 15- 08: Number of general-purpose performance monitoring counter per logical processor<br>Bits 23 - 16: Bit width of general-purpose, performance monitoring counter<br>Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events |
| | EBX | Bit 0: Core cycle event not available if 1<br>Bit 1: Instruction retired event not available if 1<br>Bit 2: Reference cycles event not available if 1<br>Bit 3: Last-level cache reference event not available if 1<br>Bit 4: Last-level cache misses event not available if 1<br>Bit 5: Branch instruction retired event not available if 1<br>Bit 6: Branch mispredict retired event not available if 1<br>Bits 31- 07: Reserved = 0 |
| | ECX | Reserved = 0 |
| | EDX | Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1)<br>Bits 12- 05: Bit width of fixed-function performance counters (if Version ID > 1)<br>Reserved = 0 |
| | *Extended Function CPUID Information* | |
| 80000000H | EAX | Maximum Input Value for Extended Function CPUID Information (see Table 3-13). |
| | EBX | Reserved |
| | ECX | Reserved |
| | EDX | Reserved |
| 80000001H | EAX | Extended Processor Signature and Extended Feature Bits. |
| | EBX | Reserved |
| | ECX | Bit 0: LAHF/SAHF available in 64-bit mode<br>Bits 31-1 Reserved |

**Table 3-12. Information Returned by CPUID Instruction (Contd.)**

| Initial EAX Value | Information Provided about the Processor | |
|---|---|---|
| | EDX | Bits 10-0: Reserved<br>Bit 11: SYSCALL/SYSRET available (when in 64-bit mode)<br>Bits 19-12: Reserved = 0<br>Bit 20: Execute Disable Bit available<br>Bits 28-21: Reserved = 0<br>Bit 29: Intel® 64 Technology available = 1<br>Bits 31-30: Reserved = 0 |
| 80000002H | EAX<br>EBX<br>ECX<br>EDX | Processor Brand String<br>Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued |
| 80000003H | EAX<br>EBX<br>ECX<br>EDX | Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued |
| 80000004H | EAX<br>EBX<br>ECX<br>EDX | Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued<br>Processor Brand String Continued |
| 80000005H | EAX<br>EBX<br>ECX<br>EDX | Reserved = 0<br>Reserved = 0<br>Reserved = 0<br>Reserved = 0 |
| 80000006H | EAX<br>EBX<br>ECX<br><br><br>EDX | Reserved = 0<br>Reserved = 0<br>Bits 7-0: Cache Line size in bytes<br>Bits 15-12: L2 Associativity field *<br>Bits 31-16: Cache size in 1K units<br>Reserved = 0<br><br>**NOTES:**<br>* L2 associativity field encodings:<br>00H - Disabled<br>01H - Direct mapped<br>02H - 2-way<br>04H - 4-way<br>06H - 8-way<br>08H - 16-way<br>0FH - Fully associative |

**Table 3-12. Information Returned by CPUID Instruction (Contd.)**

| Initial EAX Value | Information Provided about the Processor | |
|---|---|---|
| 80000007H | EAX<br>EBX<br>ECX<br>EDX | Reserved = 0<br>Reserved = 0<br>Reserved = 0<br>Reserved = 0 |
| 80000008H | EAX | Virtual/Physical Address size<br>Bits 7-0: #Physical Address Bits*<br>Bits 15-8: #Virtual Address Bits<br>Bits 31-16: Reserved = 0 |
| | EBX<br>ECX<br>EDX | Reserved = 0<br>Reserved = 0<br>Reserved = 0 |
| | | **NOTES:**<br>* If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field. |

### INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register (see Table 3-13) and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

EBX ←756e6547h (* "Genu", with G in the low nibble of BL *)
EDX ←49656e69h (* "inel", with i in the low nibble of DL *)
ECX ←6c65746eh (* "ntel", with n in the low nibble of CL *)

### INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information

When CPUID executes with EAX set to 0, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register (see Table 3-13) and is processor specific.

### Table 3-13. Highest CPUID Source Operand for Intel 64 and IA-32 Processors

| Intel 64 or IA-32 Processors | Highest Value in EAX | |
|---|---|---|
| | Basic Information | Extended Function Information |
| Earlier Intel486 Processors | CPUID Not Implemented | CPUID Not Implemented |
| Later Intel486 Processors and Pentium Processors | 01H | Not Implemented |
| Pentium Pro and Pentium II Processors, Intel® Celeron® Processors | 02H | Not Implemented |
| Pentium III Processors | 03H | Not Implemented |
| Pentium 4 Processors | 02H | 80000004H |
| Intel Xeon Processors | 02H | 80000004H |
| Pentium M Processor | 02H | 80000004H |
| Pentium 4 Processor supporting Hyper-Threading Technology | 05H | 80000008H |
| Pentium D Processor (8xx) | 05H | 80000008H |
| Pentium D Processor (9xx) | 06H | 80000008H |
| Intel Core Duo Processor | 0AH | 80000008H |
| Intel Core 2 Duo Processor | 0AH | 80000008H |
| Intel Xeon Processor 3000, 5100, 5300 Series | 0AH | 80000008H |

## IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

## INPUT EAX = 1: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 1, version information is returned in EAX (see Figure 3-5). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 3-14 for available processor type values. Stepping IDs are provided as needed.

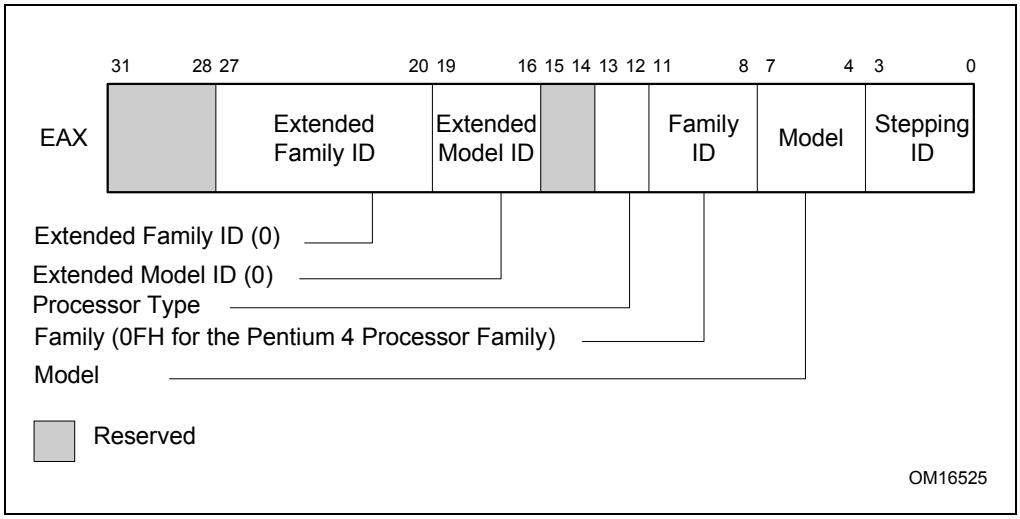



**Figure 3-5. Version Information Returned by CPUID in EAX**

**Table 3-14. Processor Type Field**

| Type | Encoding |
|---|---|
| Original OEM Processor | 00B |
| Intel OverDrive® Processor | 01B |
| Dual processor (not applicable to Intel486 processors) | 10B |
| Intel reserved | 11B |

**NOTE**

See *AP-485, Intel Processor Identification and the CPUID Instruction* (Order Number 241618) and Chapter 14 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
    THEN Displayed_Family = Family_ID;
    ELSE Displayed_Family = Extended_Family_ID + Family_ID;
```

```
        (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show Display_Family as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
    THEN Displayed_Model = (Extended_Model_ID << 4) + Model_ID;
    (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field.*)
    ELSE Displayed_Model = Model_ID;
FI;
(* Show Display_Model as HEX field. *)
```

### INPUT EAX = 1: Returns Additional Information in EBX

When CPUID executes with EAX set to 1, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed with CLFLUSH instruction in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

### INPUT EAX = 1: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 1, feature information is returned in ECX and EDX.

- Figure 3-6 and Table 3-15 show encodings for ECX.
- Figure 3-7 and Table 3-16 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

#### NOTE

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.

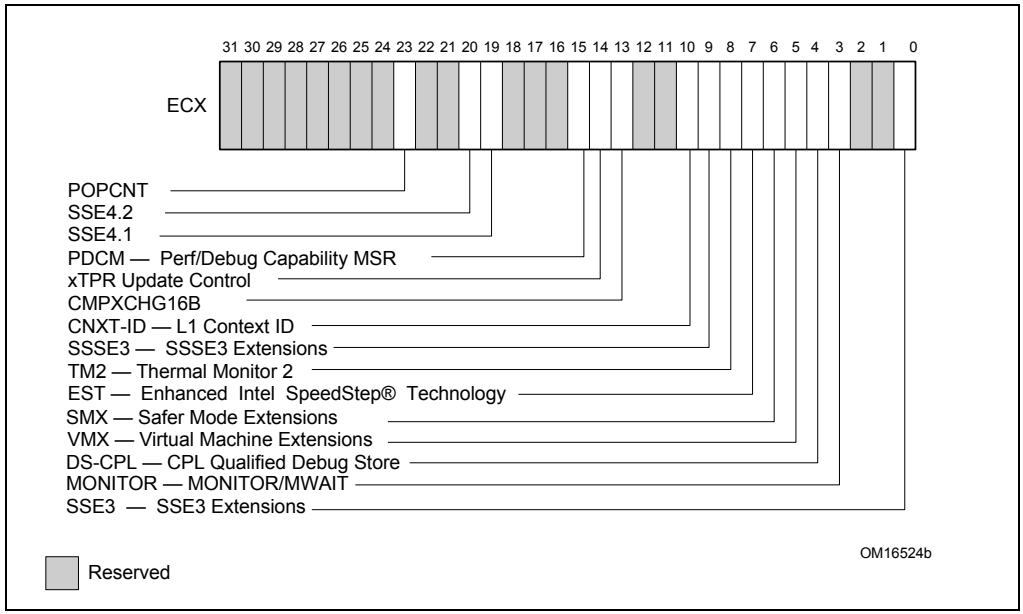


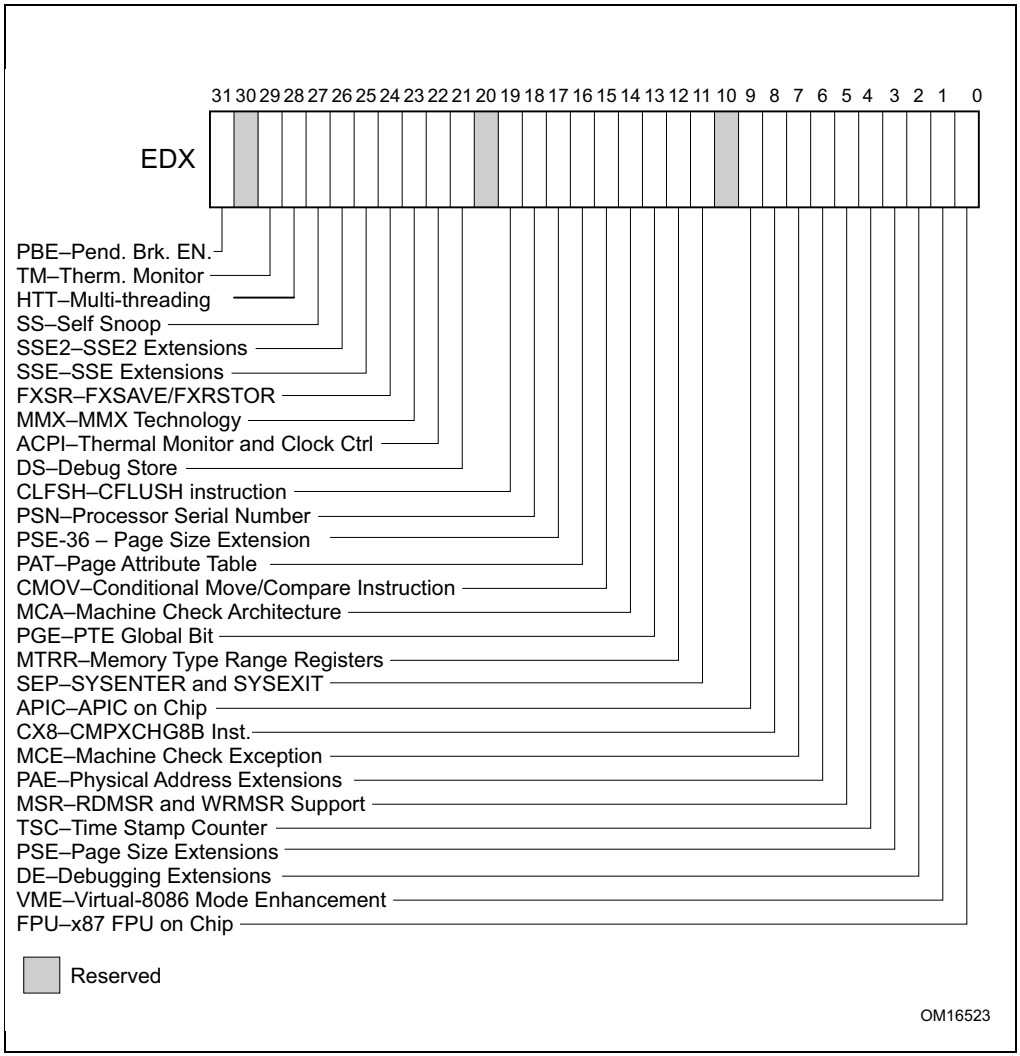

**Figure 3-6. Feature Information Returned in the ECX Register**

### Table 3-15. Feature Information Returned in the ECX Register

| Bit # | Mnemonic | Description |
|---|---|---|
| 0 | SSE3 | **Streaming SIMD Extensions 3 (SSE3)**. A value of 1 indicates the processor supports this technology. |
| 1-2 | Reserved | Reserved |
| 3 | MONITOR | **MONITOR/MWAIT**. A value of 1 indicates the processor supports this feature. |
| 4 | DS-CPL | **CPL Qualified Debug Store**. A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL. |
| 5 | VMX | Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology |
| 6 | SMX | Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 6, "Procedure Calls, Interrupts, and Exceptions". |
| 7 | EST | **Enhanced Intel SpeedStep® technology**. A value of 1 indicates that the processor supports this technology. |
| 8 | TM2 | **Thermal Monitor 2**. A value of 1 indicates whether the processor supports this technology. |
| 9 | SSSE3 | A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor |
| 10 | CNXT-ID | **L1 Context ID**. A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details. |
| 11-12 | Reserved | Reserved |
| 13 | CMPXCHG16B | **CMPXCHG16B Available**. A value of 1 indicates that the feature is available. See the "CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes" section in this chapter for a description. |
| 14 | xTPR Update Control | **xTPR Update Control**. A value of 1 indicates that the processor supports changing IA32_MISC_ENABLES[bit 23]. |
| 15 | PDCM | **Perfmon and Debug Capability**: A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES. |
| 18 - 16 | Reserved | Reserved |
| 19 | SSE4.1 | A value of 1 indicates that the processor supports SSE4.1. |
| 20 | SSE4.2 | A value of 1 indicates that the processor supports SSE4.2. |

## Table 3-15. Feature Information Returned in the ECX Register (Contd.)

| Bit # | Mnemonic | Description |
|---|---|---|
| 21 - 22 | Reserved | Reserved |
| 23 | POPCNT | A value of 1 indicates that the processor supports the POPCNT instruction. |
| 31 - 24 | Reserved | Reserved |




## Figure 3-7. Feature Information Returned in the EDX Register

### Table 3-16. More on Feature Information Returned in the EDX Register

| Bit # | Mnemonic | Description |
|---|---|---|
| 0 | FPU | **Floating Point Unit On-Chip.** The processor contains an x87 FPU. |
| 1 | VME | **Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags. |
| 2 | DE | **Debugging Extensions.** Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5. |
| 3 | PSE | **Page Size Extension.** Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs. |
| 4 | TSC | **Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege. |
| 5 | MSR | **Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent. |
| 6 | PAE | **Physical Address Extension.** Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1. The actual number of address bits beyond 32 is not defined, and is implementation specific. |
| 7 | MCE | **Machine Check Exception**. Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature. |
| 8 | CX8 | **CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic). |
| 9 | APIC | **APIC On-Chip.** The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated). |
| 10 | Reserved | Reserved |
| 11 | SEP | **SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported. |
| 12 | MTRR | **Memory Type Range Registers.** MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported. |

**Table 3-16. More on Feature Information Returned in the EDX Register (Contd.)**

| Bit # | Mnemonic | Description |
|---|---|---|
| 13 | PGE | **PTE Global Bit.** The global bit in page directory entries (PDEs) and page table entries (PTEs) is supported, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature. |
| 14 | MCA | **Machine Check Architecture.** The Machine Check Architecture, which provides a compatible mechanism for error reporting in P6 family, Pentium 4, Intel Xeon processors, and future processors, is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported. |
| 15 | CMOV | **Conditional Move Instructions.** The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported |
| 16 | PAT | **Page Attribute Table.** Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory on a 4K granularity through a linear address. |
| 17 | PSE-36 | **36-Bit Page Size Extension.** Extended 4-MByte pages that are capable of addressing physical memory beyond 4 GBytes are supported. This feature indicates that the upper four bits of the physical address of the 4-MByte page is encoded by bits 13-16 of the page directory entry. |
| 18 | PSN | **Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled. |
| 19 | CLFSH | **CLFLUSH Instruction.** CLFLUSH Instruction is supported. |
| 20 | Reserved | Reserved |
| 21 | DS | **Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and precise event-based sampling (PEBS) facilities (see Chapter 18, "Debugging and Performance Monitoring," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*). |
| 22 | ACPI | **Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control. |
| 23 | MMX | **Intel MMX Technology.** The processor supports the Intel MMX technology. |
| 24 | FXSR | **FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions. |

**Table 3-16. More on Feature Information Returned in the EDX Register (Contd.)**

| Bit # | Mnemonic | Description |
|---|---|---|
| 25 | SSE | **SSE.** The processor supports the SSE extensions. |
| 26 | SSE2 | **SSE2.** The processor supports the SSE2 extensions. |
| 27 | SS | **Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus. |
| 28 | HTT | **Multi-Threading.** The physical processor package is capable of supporting more than one logical processor. |
| 29 | TM | **Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC). |
| 30 | Reserved | Reserved |
| 31 | PBE | **Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability. |

### INPUT EAX = 2: Cache and TLB Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 2, the processor returns information about the processor's internal caches and TLBs in the EAX, EBX, ECX, and EDX registers.

The encoding is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs. The first member of the family of Pentium 4 processors will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 3-17 shows the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache or TLB types. The descriptors may appear in any order.

### Table 3-17. Encoding of Cache and TLB Descriptors

| Descriptor Value | Cache or TLB Description |
|---|---|
| 00H | Null descriptor |
| 01H | Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries |
| 02H | Instruction TLB: 4 MByte pages, 4-way set associative, 2 entries |
| 03H | Data TLB: 4 KByte pages, 4-way set associative, 64 entries |
| 04H | Data TLB: 4 MByte pages, 4-way set associative, 8 entries |
| 05H | Data TLB1: 4 MByte pages, 4-way set associative, 32 entries |
| 06H | 1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size |
| 08H | 1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size |
| 0AH | 1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size |
| 0BH | Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries |
| 0CH | 1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size |
| 22H | 3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector |
| 23H | 3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector |
| 25H | 3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector |
| 29H | 3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector |
| 2CH | 1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size |
| 30H | 1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size |
| 40H | No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache |
| 41H | 2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size |
| 42H | 2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size |
| 43H | 2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size |
| 44H | 2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size |
| 45H | 2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size |
| 46H | 3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size |
| 47H | 3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size |
| 49H | 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size |
| 50H | Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries |
| 51H | Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries |
| 52H | Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries |

### Table 3-17. Encoding of Cache and TLB Descriptors (Contd.)

| Descriptor Value | Cache or TLB Description |
|---|---|
| 56H | Data TLB0: 4 MByte pages, 4-way set associative, 16 entries |
| 57H | Data TLB0: 4 KByte pages, 4-way associative, 16 entries |
| 5BH | Data TLB: 4 KByte and 4 MByte pages, 64 entries |
| 5CH | Data TLB: 4 KByte and 4 MByte pages,128 entries |
| 5DH | Data TLB: 4 KByte and 4 MByte pages,256 entries |
| 60H | 1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size |
| 66H | 1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size |
| 67H | 1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size |
| 68H | 1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size |
| 70H | Trace cache: 12 K-µop, 8-way set associative |
| 71H | Trace cache: 16 K-µop, 8-way set associative |
| 72H | Trace cache: 32 K-µop, 8-way set associative |
| 78H | 2nd-level cache: 1 MByte, 4-way set associative, 64byte line size |
| 79H | 2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector |
| 7AH | 2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector |
| 7BH | 2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector |
| 7CH | 2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector |
| 7DH | 2nd-level cache: 2 MByte, 8-way set associative, 64byte line size |
| 7FH | 2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size |
| 82H | 2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size |
| 83H | 2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size |
| 84H | 2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size |
| 85H | 2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size |
| 86H | 2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size |
| 87H | 2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size |
| B0H | Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries |
| B3H | Data TLB: 4 KByte pages, 4-way set associative, 128 entries |
| B4H | Data TLB1: 4 KByte pages, 4-way associative, 256 entries |
| F0H | 64-Byte prefetching |
| F1H | 128-Byte prefetching |

### Example 3-1. Example of Cache and TLB Interpretation

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

```
EAX     66 5B 50 01H
EBX     0H
ECX     0H
EDX     00 7A 70 00H
```

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This indicates that CPUID needs to be executed once with an input value of 2 to retrieve complete information about caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K-μop, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sectored, 64-byte cache line size.
  - 00H - NULL descriptor.

### INPUT EAX = 4: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 4 and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-12.

The CPUID leaf 4 also reports information about maximum number of cores in a physical package. This information is constant for all valid index values. Software can

query maximum number of cores per physical package by executing CPUID with EAX=4 and ECX=0.

### INPUT EAX = 5: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 5, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-12.

### INPUT EAX = 6: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 6, the processor returns information about thermal and power management features. See Table 3-12.

### INPUT EAX = 10: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 10, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-12) is greater than Pn 0. See Table 3-12.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 18, “Debugging and Performance Monitoring,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

## METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method; this method also returns the processor’s maximum operating frequency
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: “Identification of Earlier IA-32 Processors” in Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

### The Processor Brand String Method

Figure 3-8 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the maximum operating frequency of the processor to the EAX, EBX, ECX, and EDX registers.

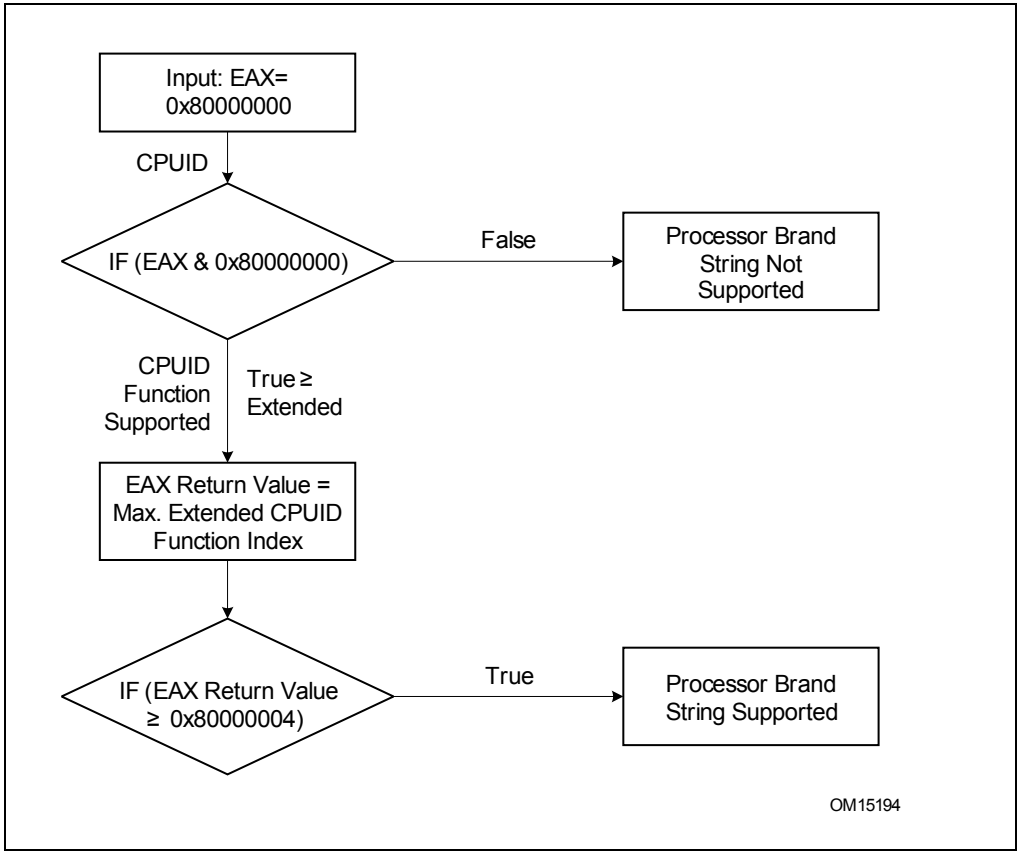



**Figure 3-8. Determination of Support for the Processor Brand String**

## How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 8000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

Table 3-18 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

**Table 3-18. Processor Brand String Returned with Pentium 4 Processor**

| EAX Input Value | Return Values | ASCII Equivalent |
|---|---|---|
| 80000002H | EAX = 20202020H<br>EBX = 20202020H<br>ECX = 20202020H<br>EDX = 6E492020H | "  "<br>"  "<br>"  "<br>"nI  " |
| 80000003H | EAX = 286C6574H<br>EBX = 50202952H<br>ECX = 69746E65H<br>EDX = 52286D75H | "(let"<br>"P )R"<br>"itne"<br>"R(mu" |
| 80000004H | EAX = 20342029H<br>EBX = 20555043H<br>ECX = 30303531H<br>EDX = 007A484DH | " 4 )"<br>" UPC"<br>"0051"<br>"\0zHM" |

## Extracting the Maximum Processor Frequency from Brand Strings

Figure 3-9 provides an algorithm which software can use to extract the maximum processor operating frequency from the processor brand string.

### NOTE

When a frequency is given in a brand string, it is the maximum qualified frequency of the processor, not the frequency at which the processor is currently running.

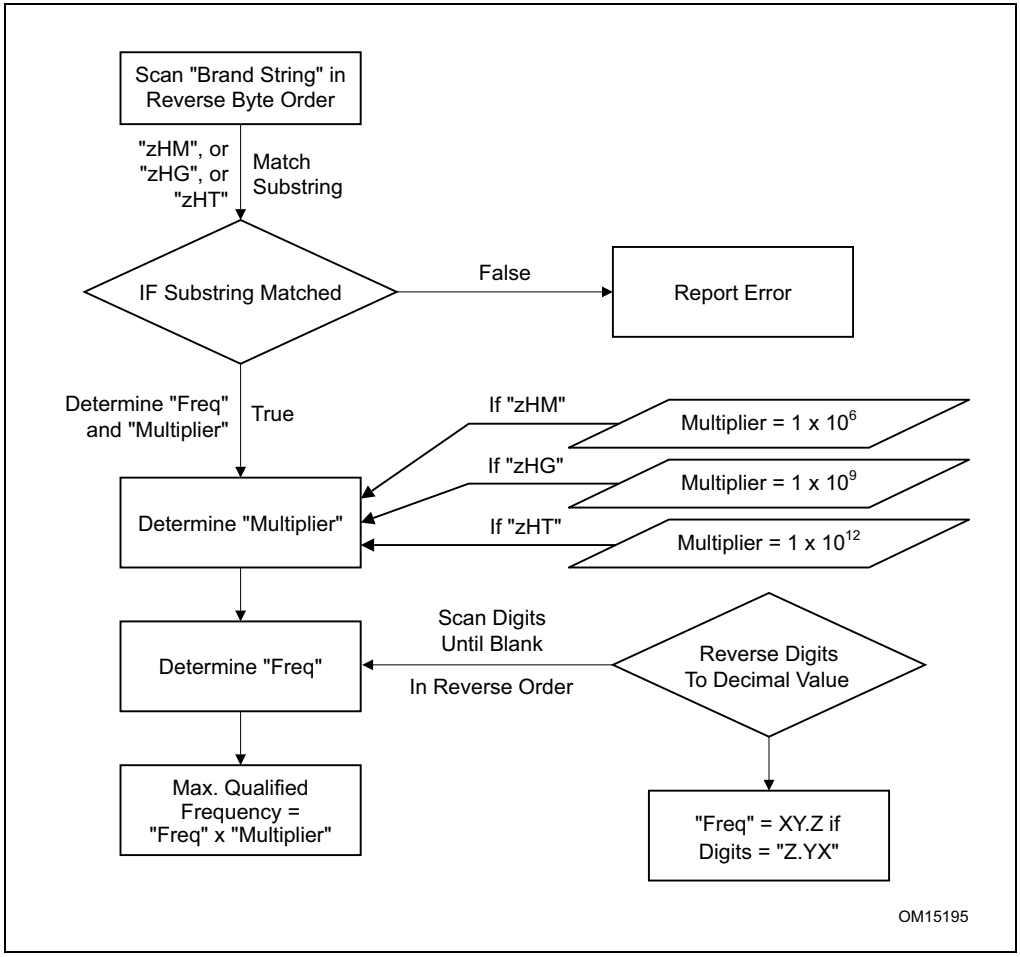



**Figure 3-9. Algorithm for Extracting Maximum Processor Frequency**

## The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associate with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that

do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-19 shows brand indices that have identification strings associated with them.

**Table 3-19. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings**

| Brand Index | Brand String |
|---|---|
| 00H | This processor does not support the brand identification feature |
| 01H | Intel(R) Celeron(R) processor[1] |
| 02H | Intel(R) Pentium(R) III processor[1] |
| 03H | Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor |
| 04H | Intel(R) Pentium(R) III processor |
| 06H | Mobile Intel(R) Pentium(R) III processor-M |
| 07H | Mobile Intel(R) Celeron(R) processor[1] |
| 08H | Intel(R) Pentium(R) 4 processor |
| 09H | Intel(R) Pentium(R) 4 processor |
| 0AH | Intel(R) Celeron(R) processor[1] |
| 0BH | Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP |
| 0CH | Intel(R) Xeon(R) processor MP |
| 0EH | Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor |
| 0FH | Mobile Intel(R) Celeron(R) processor[1] |
| 11H | Mobile Genuine Intel(R) processor |
| 12H | Intel(R) Celeron(R) M processor |
| 13H | Mobile Intel(R) Celeron(R) processor[1] |
| 14H | Intel(R) Celeron(R) processor |
| 15H | Mobile Genuine Intel(R) processor |
| 16H | Intel(R) Pentium(R) M processor |
| 17H | Mobile Intel(R) Celeron(R) processor[1] |
| 18H – 0FFH | RESERVED |

NOTES:

1. Indicates versions of these processors that were introduced after the Pentium III

## IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

## Operation

```
IA32_BIOS_SIGN_ID MSR ←Update with installed microcode revision number;

CASE (EAX) OF
    EAX = 0:
        EAX ←Highest basic function input value understood by CPUID;
        EBX ←Vendor identification string;
        EDX ←Vendor identification string;
        ECX ←Vendor identification string;
    BREAK;
    EAX = 1H:
        EAX[3:0] ←Stepping ID;
        EAX[7:4] ←Model;
        EAX[11:8] ←Family;
        EAX[13:12] ←Processor type;
        EAX[15:14] ←Reserved;
        EAX[19:16] ←Extended Model;
        EAX[23:20] ←Extended Family;
        EAX[31:24] ←Reserved;
        EBX[7:0] ←Brand Index; (* Reserved if the value is zero. *)
        EBX[15:8] ←CLFLUSH Line Size;
        EBX[16:23] ←Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
        EBX[24:31] ←Initial APIC ID;
        ECX ←Feature flags; (* See Figure 3-6. *)
        EDX ←Feature flags; (* See Figure 3-7. *)
    BREAK;
    EAX = 2H:
        EAX ←Cache and TLB information;
        EBX ←Cache and TLB information;
        ECX ←Cache and TLB information;
        EDX ←Cache and TLB information;
    BREAK;
    EAX = 3H:
        EAX ←Reserved;
        EBX ←Reserved;
        ECX ←ProcessorSerialNumber[31:0];
        (* Pentium III processors only, otherwise reserved. *)
        EDX ←ProcessorSerialNumber[63:32];
        (* Pentium III processors only, otherwise reserved. *
```

```
BREAK
EAX = 4H:
    EAX ←Deterministic Cache Parameters Leaf; (* See Table 3-12. *)
    EBX ←Deterministic Cache Parameters Leaf;
    ECX ←Deterministic Cache Parameters Leaf;
    EDX ←Deterministic Cache Parameters Leaf;
BREAK;
EAX = 5H:
    EAX ←MONITOR/MWAIT Leaf; (* See Table 3-12. *)
    EBX ←MONITOR/MWAIT Leaf;
    ECX ←MONITOR/MWAIT Leaf;
    EDX ←MONITOR/MWAIT Leaf;
BREAK;
EAX = 6H:
    EAX ←Thermal and Power Management Leaf; (* See Table 3-12. *)
    EBX ←Thermal and Power Management Leaf;
    ECX ←Thermal and Power Management Leaf;
    EDX ←Thermal and Power Management Leaf;
BREAK;
EAX = 7H or 8H or 9H:
    EAX ←Reserved = 0;
    EBX ←Reserved = 0;
    ECX ←Reserved = 0;
    EDX ←Reserved = 0;
BREAK;
EAX = AH:
    EAX ←Architectural Performance Monitoring Leaf; (* See Table 3-12. *)
    EBX ←Architectural Performance Monitoring Leaf;
    ECX ←Architectural Performance Monitoring Leaf;
    EDX ←Architectural Performance Monitoring Leaf;
BREAK;
EAX = 80000000H:
    EAX ←Highest extended function input value understood by CPUID;
    EBX ←Reserved;
    ECX ←Reserved;
    EDX ←Reserved;
BREAK;
EAX = 80000001H:
    EAX ←Reserved;
    EBX ←Reserved;
    ECX ←Extended Feature Bits (* See Table 3-12.*);
    EDX ←Extended Feature Bits (* See Table 3-12. *);
BREAK;
```

```
    EAX = 80000002H:
        EAX ←Processor Brand String;
        EBX ←Processor Brand String, continued;
        ECX ←Processor Brand String, continued;
        EDX ←Processor Brand String, continued;
    BREAK;
    EAX = 80000003H:
        EAX ←Processor Brand String, continued;
        EBX ←Processor Brand String, continued;
        ECX ←Processor Brand String, continued;
        EDX ←Processor Brand String, continued;
    BREAK;
    EAX = 80000004H:
        EAX ←Processor Brand String, continued;
        EBX ←Processor Brand String, continued;
        ECX ←Processor Brand String, continued;
        EDX ←Processor Brand String, continued;
    BREAK;
    EAX = 80000005H:
        EAX ←Reserved = 0;
        EBX ←Reserved = 0;
        ECX ←Reserved = 0;
        EDX ←Reserved = 0;
    BREAK;
    EAX = 80000006H:
        EAX ←Reserved = 0;
        EBX ←Reserved = 0;
        ECX ←Cache information;
        EDX ←Reserved = 0;
    BREAK;
    EAX = 80000007H:
        EAX ←Reserved = 0;
        EBX ←Reserved = 0;
        ECX ←Reserved = 0;
        EDX ←Reserved = 0;
    BREAK;
    EAX = 80000008H:
        EAX ←Reserved = 0;
        EBX ←Reserved = 0;
        ECX ←Reserved = 0;
        EDX ←Reserved = 0;
    BREAK;
    DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
```

```
        EAX ←Reserved; (* Information returned for highest basic information leaf. *)
        EBX ←Reserved; (* Information returned for highest basic information leaf. *)
        ECX ←Reserved; (* Information returned for highest basic information leaf. *)
        EDX ←Reserved; (* Information returned for highest basic information leaf. *)
    BREAK;
ESAC;
```

## Flags Affected

None.

## Exceptions (All Operating Modes)

| | |
|---|---|
| #UD | If the LOCK prefix is used.<br>In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated. |

# CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F E6 | CVTDQ2PD *xmm1, xmm2/m64* | Valid | Valid | Convert two packed signed doubleword integers from *xmm2/m128* to two packed double-precision floating-point values in *xmm1*. |

## Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the packed integers are located in the low quadword of the register.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[127:64] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTDQ2PD __m128d _mm_cvtepi32_pd(__m128i a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |

| | |
|---|---|
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 5B /r | CVTDQ2PS *xmm1, xmm2/m128* | Valid | Valid | Convert four packed signed doubleword integers from *xmm2/m128* to four packed single-precision floating-point values in *xmm1*. |

## Description

Converts four packed signed doubleword integers in the source operand (second operand) to four packed single-precision floating-point values in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. When a conversion is inexact, rounding is performed according to the rounding control bits in the MXCSR register.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
DEST[95:64] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64]);
DEST[127:96] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTDQ2PS __m128 _mm_cvtepi32_ps(__m128i a)

## SIMD Floating-Point Exceptions

Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F E6 | CVTPD2DQ *xmm1, xmm2/m128* | Valid | Valid | Convert two packed double-precision floating-point values from *xmm2/m128* to two packed signed doubleword integers in *xmm1*. |

## Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand and the high quadword is cleared to all 0s.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64]);
DEST[127:64] ←0000000000000000H;
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2DQ __m128i _mm_cvtpd_epi32(__m128d a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.segments. |

| | |
|---|---|
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |

| | |
|---|---|
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 2D /r | CVTPD2PI *mm, xmm/m128* | Valid | Valid | Convert two packed double-precision floating-point values from *xmm/m128* to two packed signed doubleword integers in *mm*. |

## Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTPD1PI __m64 _mm_cvtpd_pi32(__m128d a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #MF | If there is a pending x87 FPU exception. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form.<br>If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #MF | If there is a pending x87 FPU exception. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 5A /r | CVTPD2PS *xmm1, xmm2/m128* | Valid | Valid | Convert two packed double-precision floating-point values in *xmm2/m128* to two packed single-precision floating-point values in *xmm1*. |

## Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand, and the high quadword is cleared to all 0s. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
DEST[63:32] ←Convert_Double_Precision_To_Single_Precision_
              Floating_Point(SRC[127:64]);
DEST[127:64] ←0000000000000000H;
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2PS __m128 _mm_cvtpd_ps(__m128d a)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |

| | |
|---|---|
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 2A /r | CVTPI2PD *xmm, mm/m64** | Valid | Valid | Convert two packed signed doubleword integers from *mm/mem64* to two packed double-precision floating-point values in *xmm*. |

**NOTES:**

* Operation is different for different operand sets; see the Description section.

## Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. In addition, depending on the operand configuration:

- **For operands *xmm*, *mm*:** the instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PD instruction is executed.
- **For operands *xmm*, *m64*:** the instruction does not cause a transition to MMX technology and does not take x87 FPU exceptions.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[127:64] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PD __m128d _mm_cvtpi32_pd(__m64 a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #MF | If there is a pending x87 FPU exception. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 2A /*r* | CVTPI2PS *xmm, mm/m64* | Valid | Valid | Convert two signed doubleword integers from *mm/m64* to two single-precision floating-point values in *xmm*. |

## Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
(* High quadword of destination unchanged *)
```

## Intel C/C++Compiler Intrinsic Equivalent

CVTPI2PS __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)

## SIMD Floating-Point Exceptions

Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 5B /r | CVTPS2DQ *xmm1, xmm2/m128* | Valid | Valid | Convert four packed single-precision floating-point values from *xmm2/m128* to four packed signed doubleword integers in *xmm1*. |

## Description

Converts four packed single-precision floating-point values in the source operand (second operand) to four packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);
DEST[95:64] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64]);
DEST[127:96] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2DQ __m128i _mm_cvtps_epi32(__m128 a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |

| | |
|---|---|
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |

| | |
|---|---|
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 5A /r | CVTPS2PD *xmm1, xmm2/m64* | Valid | Valid | Convert two packed single-precision floating-point values in *xmm2/m64* to two packed double-precision floating-point values in *xmm1*. |

## Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the packed single-precision floating-point values are contained in the low quadword of the register.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

DEST[63:0] ←Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[127:64] ←Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32]);

## Intel C/C++Compiler Intrinsic Equivalent

CVTPS2PD __m128d _mm_cvtps_pd(__m128 a)

## SIMD Floating-Point Exceptions

Invalid, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 2D /r | CVTPS2PI *mm, xmm/m64* | Valid | Valid | Convert two packed single-precision floating-point values from *xmm/m64* to two packed signed doubleword integers in *mm*. |

## Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

CVTPS2PI causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PI __m64 _mm_cvtps_pi32(__m128 a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #MF | If there is a pending x87 FPU exception. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 2D /r | CVTSD2SI *r32, xmm/m64* | Valid | Valid | Convert one double-precision floating-point value from *xmm/m64* to one signed doubleword integer *r32*. |
| F2 REX.W 0F 2D /r | CVTSD2SI *r64, xmm/m64* | Valid | N.E. | Convert one double-precision floating-point value from *xmm/m64* to one signed quadword integer sign-extended into *r64*. |

## Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
    ELSE
        DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0]);
FI;
```

## Intel C/C++Compiler Intrinsic Equivalent

int _mm_cvtsd_si32(__m128d a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 5A /r | CVTSD2SS *xmm1, xmm2/m64* | Valid | Valid | Convert one double-precision floating-point value in *xmm2/m64* to one single-precision floating-point value in *xmm1*. |

## Description

Converts a double-precision floating-point value in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand).

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand, and the upper 3 doublewords are left unchanged. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0]);
(* DEST[127:32] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTSD2SS __m128 _mm_cvtsd_ss(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 2A /r | CVTSI2SD *xmm, r/m32* | Valid | Valid | Convert one signed doubleword integer from *r/m32* to one double-precision floating-point value in *xmm*. |
| F2 REX.W 0F 2A /r | CVTSI2SD *xmm, r/m64* | Valid | N.E. | Convert one signed quadword integer from *r/m64* to one double-precision floating-point value in *xmm*. |

## Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the source operand (second operand) to a double-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);
        (* DEST[127:64] unchanged *)
    ELSE
        DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
        (* DEST[127:64] unchanged *)
FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SD __m128d _mm_cvtsi32_sd(__m128d a, int b)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 2A /r | CVTSI2SS *xmm, r/m32* | Valid | Valid | Convert one signed doubleword integer from *r/m32* to one single-precision floating-point value in *xmm*. |
| F3 REX.W 0F 2A /r | CVTSI2SS *xmm, r/m64* | Valid | N.E. | Convert one signed quadword integer from *r/m64* to one single-precision floating-point value in *xmm*. |

## Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
        (* DEST[127:32] unchanged *)
    ELSE
        DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
        (* DEST[127:32] unchanged *)
FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SS __m128 _mm_cvtsi32_ss(__m128 a, int b)

## SIMD Floating-Point Exceptions

Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 5A /r | CVTSS2SD *xmm1, xmm2/m32* | Valid | Valid | Convert one single-precision floating-point value in *xmm2/m32* to one double-precision floating-point value in *xmm1*. |

## Description

Converts a single-precision floating-point value in the source operand (second operand) to a double-precision floating-point value in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand, and the high quadword is left unchanged.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
(* DEST[127:64] unchanged *)
```

## Intel C/C++Compiler Intrinsic Equivalent

CVTSS2SD __m128d _mm_cvtss_sd(__m128d a, __m128 b)

## SIMD Floating-Point Exceptions

Invalid, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 2D /r | CVTSS2SI *r32, xmm/m32* | Valid | Valid | Convert one single-precision floating-point value from *xmm/m32* to one signed doubleword integer in *r32*. |
| F3 REX.W 0F 2D /r | CVTSS2SI *r64, xmm/m32* | Valid | N.E. | Convert one single-precision floating-point value from *xmm/m32* to one signed quadword integer in *r64*. |

## Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF 64-bit Mode and OperandSize = 64
    THEN
        DEST[64:0] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
    ELSE
        DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cvtss_si32(__m128d a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 2C /r | CVTTPD2PI *mm, xmm/m128* | Valid | Valid | Convert two packer double-precision floating-point values from *xmm/m128* to two packed signed doubleword integers in *mm* using truncation. |

## Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer_
            Truncate(SRC[127:64]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD1PI__m64 _mm_cvttpd_pi32(__m128d a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |

| | |
|---|---|
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #MF | If there is a pending x87 FPU exception. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F E6 | CVTTPD2DQ *xmm1, xmm2/m128* | Valid | Valid | Convert two packed double-precision floating-point values from *xmm2/m128* to two packed signed doubleword integers in *xmm1* using truncation. |

## Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand and the high quadword is cleared to all 0s.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0]);
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer_
              Truncate(SRC[127-64]);
DEST[127:64] ←0000000000000000H;
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD2DQ __m128i _mm_cvttpd_epi32(__m128d a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |

| | |
|---|---|
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |

| | |
|---|---|
| #GP(0) | If the memory address is in a non-canonical form.<br>If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 5B /r | CVTTPS2DQ *xmm1, xmm2/m128* | Valid | Valid | Convert four single-precision floating-point values from *xmm2/m128* to four signed doubleword integers in *xmm1* using truncation. |

## Description

Converts four packed single-precision floating-point values in the source operand (second operand) to four packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);
DEST[95:64] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64]);
DEST[127:96] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2DQ __m128i _mm_cvttps_epi32(__m128 a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 2C /r | CVTTPS2PI *mm, xmm/m64* | Valid | Valid | Convert two single-precision floating-point values from *xmm/m64* to two signed doubleword signed integers in *mm* using truncation. |

## Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI __m64 _mm_cvttps_pi32(__m128 a)

## SIMD Floating-Point Exceptions

Invalid, Precision.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #MF | If there is a pending x87 FPU exception. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Doubleword Integer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 2C /*r* | CVTTSD2SI *r32, xmm/m64* | Valid | Valid | Convert one double-precision floating-point value from *xmm/m64* to one signed doubleword integer in *r32* using truncation. |
| F2 REX.W 0F 2C /*r* | CVTTSD2SI *r64, xmm/m64* | Valid | N.E. | Convert one double precision floating-point value from *xmm/m64* to one signedquadword integer in *r64* using truncation. |

## Description

Converts a double-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ←Convert_Double_Precision_Floating_Point_To_
                    Integer_Truncate(SRC[63:0]);
    ELSE
        DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_
                    Integer_Truncate(SRC[63:0]);
FI;
```

### Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cvttsd_si32(__m128d a)

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 2C /r | CVTTSS2SI *r32, xmm/m32* | Valid | Valid | Convert one single-precision floating-point value from *xmm/m32* to one signed doubleword integer in *r32* using truncation. |
| F3 REX.W 0F 2C /r | CVTTSS2SI *r64, xmm/m32* | Valid | N.E. | Convert one single-precision floating-point value from *xmm/m32* to one signed quadword integer in *r64* using truncation. |

## Description

Converts a single-precision floating-point value in the source operand (second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H) is returned.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ←Convert_Single_Precision_Floating_Point_To_
                    Integer_Truncate(SRC[31:0]);
    ELSE
        DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_
                    Integer_Truncate(SRC[31:0]);
FI;
```

### Intel C/C++Compiler Intrinsic Equivalent

int _mm_cvttss_si32(__m128d a)

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 99 | CWD | Valid | Valid | DX:AX ←sign-extend of AX. |
| 99 | CDQ | Valid | Valid | EDX:EAX ←sign-extend of EAX. |
| REX.W + 99 | CQO | Valid | N.E. | RDX:RAX←sign-extend of RAX. |

## Description

Doubles the size of the operand in register AX, EAX, or RAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX, EDX:EAX, or RDX:RAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register. The CQO instruction (available in 64-bit mode only) copies the sign (bit 63) of the value in the RAX register into every bit position in the RDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before word division. The CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CQO instruction can be used to produce a double quadword dividend from a quadword before a quadword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. The CQO mnemonics reference the same opcode as CWD/CDQ. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF OperandSize = 16 (* CWD instruction *)
    THEN
        DX ←SignExtend(AX);
    ELSE IF OperandSize = 32 (* CDQ instruction *)
        EDX ←SignExtend(EAX); FI;
    ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
```

```
        RDX ←SignExtend(RAX); FI;
FI;
```

## Flags Affected

None.

## Exceptions (All Operating Modes)

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

# DAA—Decimal Adjust AL after Addition

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 27 | DAA | Invalid | Valid | Decimal adjust AL after addition. |

## Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

## Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        old_AL ←AL;
        old_CF ←CF;
        CF ←0;
        IF (((AL AND 0FH) > 9) or AF = 1)
            THEN
                AL ←AL +6;
                CF ←old_CF or (Carry from AL ←AL +6);
                AF ←1;
            ELSE
                AF ←0;
        FI;
        IF ((old_AL > 99H) or (old_CF = 1))
            THEN
                AL ←AL +60H;
                CF ←1;
            ELSE
                CF ←0;
        FI;
FI;
```

## Example

```
ADD     AL, BL    Before: AL=79H  BL=35H EFLAGS(OSZAPC)=XXXXXX
                  After:  AL=AEH  BL=35H EFLAGS(OSZAPC)=110000
DAA               Before: AL=AEH  BL=35H EFLAGS(OSZAPC)=110000
                  After:  AL=14H  BL=35H EFLAGS(OSZAPC)=X00111
DAA               Before: AL=2EH  BL=35H EFLAGS(OSZAPC)=110000
                  After:  AL=34H  BL=35H EFLAGS(OSZAPC)=X00101
```

## Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see the “Operation” section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## Compatibility Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If in 64-bit mode. |

# DAS—Decimal Adjust AL after Subtraction

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 2F | DAS | Invalid | Valid | Decimal adjust AL after subtraction. |

## Description

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

## Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        old_AL ←AL;
        old_CF ←CF;
        CF ←0;
        IF (((AL AND 0FH) > 9) or AF = 1)
            THEN
                AL ←AL –6;
                CF ←old_CF or (Borrow from AL ←AL –6);
                AF ←1;
            ELSE
                AF ←0;
        FI;
        IF ((old_AL > 99H) or (old_CF = 1))
            THEN
                AL ←AL –60H;
                CF ←1;
        FI;
FI;
```

## Example

```
SUB     AL, BL    Before: AL = 35H, BL = 47H, EFLAGS(OSZAPC) = XXXXXX
                  After:  AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
DAA               Before: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
                  After:  AL = 88H, BL = 47H, EFLAGS(OSZAPC) = X10111
```

## Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal borrow in either digit of the result (see the “Operation” section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## Compatibility Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If in 64-bit mode. |

# DEC—Decrement by 1

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| FE /1 | DEC *r/m8* | Valid | Valid | Decrement *r/m8* by 1. |
| REX + FE /1 | DEC *r/m8** | Valid | N.E. | Decrement *r/m8* by 1. |
| FF /1 | DEC *r/m16* | Valid | Valid | Decrement *r/m16* by 1. |
| FF /1 | DEC *r/m32* | Valid | Valid | Decrement *r/m32* by 1. |
| REX.W + FF /1 | DEC *r/m64* | Valid | N.E. | Decrement *r/m64* by 1. |
| 48+rw | DEC *r16* | N.E. | Valid | Decrement *r16* by 1. |
| 48+rd | DEC *r32* | N.E. | Valid | Decrement *r32* by 1. |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, DEC r16 and DEC r32 are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

## Operation

DEST ←DEST – 1;

## Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination operand is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

# DIV—Unsigned Divide

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F6 /6 | DIV *r/m8* | Valid | Valid | Unsigned divide AX by *r/m8*, with result stored in AL ←Quotient, AH ← Remainder. |
| REX + F6 /6 | DIV *r/m8** | Valid | N.E. | Unsigned divide AX by *r/m8*, with result stored in AL ←Quotient, AH ← Remainder. |
| F7 /6 | DIV *r/m16* | Valid | Valid | Unsigned divide DX:AX by *r/m16*, with result stored in AX ←Quotient, DX ← Remainder. |
| F7 /6 | DIV *r/m32* | Valid | Valid | Unsigned divide EDX:EAX by *r/m32*, with result stored in EAX ←Quotient, EDX ←Remainder. |
| REX.W + F7 /6 | DIV *r/m64* | Valid | N.E. | Unsigned divide RDX:RAX by *r/m64*, with result stored in RAX ←Quotient, RDX ←Remainder. |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-20.

**Table 3-20. DIV Action**

| Operand Size | Dividend | Divisor | Quotient | Remainder | Maximum Quotient |
|---|---|---|---|---|---|
| Word/byte | AX | r/m8 | AL | AH | 255 |
| Doubleword/word | DX:AX | r/m16 | AX | DX | 65,535 |
| Quadword/doubleword | EDX:EAX | r/m32 | EAX | EDX | $2^{32}-1$ |
| Doublequadword/ quadword | RDX:RAX | r/m64 | RAX | RDX | $2^{64}-1$ |

## Operation

```
IF SRC = 0
    THEN #DE; FI; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
    THEN
        temp ←AX / SRC;
        IF temp > FFH
            THEN #DE; (* Divide error *)
            ELSE
                AL ←temp;
                AH ←AX MOD SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp ←DX:AX / SRC;
            IF temp > FFFFH
                THEN #DE; (* Divide error *)
            ELSE
                AX ←temp;
                DX ←DX:AX MOD SRC;
            FI;
        FI;
    ELSE IF Operandsize = 32 (* Quadword/doubleword operation *)
        THEN
            temp ←EDX:EAX / SRC;
            IF temp > FFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                EAX ←temp;
                EDX ←EDX:EAX MOD SRC;
            FI;
        FI;
```

```
    ELSE IF 64-Bit Mode and Operandsize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp ←RDX:RAX / SRC;
            IF temp > FFFFFFFFFFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                RAX ←temp;
                RDX ←RDX:RAX MOD SRC;
            FI;
        FI;
FI;
```

## Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #DE | If the source operand (divisor) is 0 |
| | If the quotient is too large for the designated register. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #DE | If the source operand (divisor) is 0. |
| | If the quotient is too large for the designated register. |
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #DE | If the source operand (divisor) is 0. |
| | If the quotient is too large for the designated register. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #DE | If the source operand (divisor) is 0 |
| | If the quotient is too large for the designated register. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# DIVPD—Divide Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 5E /r | DIVPD *xmm1, xmm2/m128* | Valid | Valid | Divide packed double-precision floating-point values in *xmm1* by packed double-precision floating-point values *xmm2/m128.* |

## Description

Performs a SIMD divide of the *two* packed double-precision floating-point values in the destination operand (first operand) by the *two* packed double-precision floating-point values in the source operand (second operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a SIMD double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←DEST[63:0] / (SRC[63:0]);
DEST[127:64] ←DEST[127:64] / (SRC[127:64]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

DIVPD __m128d _mm_div_pd(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# DIVPS—Divide Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 5E /r | DIVPS *xmm1, xmm2/m128* | Valid | Valid | Divide packed single-precision floating-point values in *xmm1* by packed single-precision floating-point values *xmm2/m128*. |

## Description

Performs a SIMD divide of the *four* packed single-precision floating-point values in the destination operand (first operand) by the *four* packed single-precision floating-point values in the source operand (second operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a SIMD single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←DEST[31:0] / (SRC[31:0]);
DEST[63:32] ←DEST[63:32] / (SRC[63:32]);
DEST[95:64] ←DEST[95:64] / (SRC[95:64]);
DEST[127:96] ←DEST[127:96] / (SRC[127:96]);
```

## Intel C/C++ Compiler Intrinsic Equivalent

DIVPS __m128 _mm_div_ps(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

# DIVSD—Divide Scalar Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 5E /r | DIVSD *xmm1, xmm2/m64* | Valid | Valid | Divide low double-precision floating-point value n *xmm1* by low double-precision floating-point value in *xmm2/mem64.* |

## Description

Divides the low double-precision floating-point value in the destination operand (first operand) by the low double-precision floating-point value in the source operand (second operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a scalar double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0]←DEST[63:0] / SRC[63:0];
(* DEST[127:64] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

DIVSD __m128d _mm_div_sd (m128d a, m128d b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |

| | |
|---|---|
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |

| | |
|---|---|
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# DIVSS—Divide Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 5E /r | DIVSS *xmm1, xmm2/m32* | Valid | Valid | Divide low single-precision floating-point value in *xmm1* by low single-precision floating-point value in *xmm2/m32.* |

## Description

Divides the low single-precision floating-point value in the destination operand (first operand) by the low single-precision floating-point value in the source operand (second operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a scalar single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0]←DEST[31:0] / SRC[31:0];
(* DEST[127:32] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

DIVSS __m128 _mm_div_ss(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |

| | |
|---|---|
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |

| | |
|---|---|
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# EMMS—Empty MMX Technology State

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 77 | EMMS | Valid | Valid | Set the x87 FPU tag word to empty. |

## Description

Sets the values of all the tags in the x87 FPU tag word to empty (all 1s). This operation marks the x87 FPU data registers (which are aliased to the MMX technology registers) as available for use by x87 FPU floating-point instructions. (See Figure 8-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the format of the x87 FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in x87 FPU tag word to valid (all 0s).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

EMMS operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
x87FPUTagWord ←FFFFH;
```

## Intel C/C++ Compiler Intrinsic Equivalent

void _mm_empty()

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# ENTER—Make Stack Frame for Procedure Parameters

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| C8 *iw* 00 | ENTER *imm16,* 0 | Valid | Valid | Create a stack frame for a procedure. |
| C8 *iw* 01 | ENTER *imm16,*1 | Valid | Valid | Create a nested stack frame for a procedure. |
| C8 *iw* ib | ENTER *imm16, imm8* | Valid | Valid | Create a nested stack frame for a procedure. |

## Description

Creates a stack frame for a procedure. The first operand (size operand) specifies the size of the stack frame (that is, the number of bytes of dynamic storage allocated on the stack for the procedure). The second operand (nesting level operand) gives the lexical nesting level (0 to 31) of the procedure. The nesting level determines the number of stack frame pointers that are copied into the “display area” of the new stack frame from the preceding frame. Both of these operands are immediate values.

The stack-size attribute determines whether the BP (16 bits), EBP (32 bits), or RBP (64 bits) register specifies the current frame pointer and whether SP (16 bits), ESP (32 bits), or RSP (64 bits) specifies the stack pointer. In 64-bit mode, stack-size attribute is always 64-bits.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See “Procedure Calls for Block-Structured Languages” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the actions of the ENTER instruction.

The ENTER instruction causes a page fault whenever a write using the final value of the stack pointer (within the current stack segment) would do so.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded.

## Operation

```
NestingLevel ←NestingLevel MOD 32
IF 64-Bit Mode (StackSize = 64)
    THEN
        Push(RBP);
        FrameTemp ←RSP;
    ELSE IF StackSize = 32
        THEN
            Push(EBP);
            FrameTemp ←ESP; FI;
    ELSE (* StackSize = 16 *)
            Push(BP);
            FrameTemp ←SP;
FI;
IF NestingLevel = 0
    THEN GOTO CONTINUE;
FI;

IF (NestingLevel > 1)
    THEN FOR i ←1 to (NestingLevel - 1)
        DO
            IF 64-Bit Mode (StackSize = 64)
                THEN
                    RBP ←RBP - 8;
                    Push([RBP]); (* Quadword push *)
                ELSE IF OperandSize = 32
                    THEN
                        IF StackSize = 32
                            EBP ←EBP - 4;
                            Push([EBP]); (* Doubleword push *)
                        ELSE (* StackSize = 16 *)
                            BP ←BP - 4;
                            Push([BP]); (* Doubleword push *)
                        FI;
                    FI;
                ELSE (* OperandSize = 16 *)
                    IF StackSize = 32
                        THEN
                            EBP ←EBP - 2;
                            Push([EBP]); (* Word push *)
                        ELSE (* StackSize = 16 *)
                            BP ←BP - 2;
                            Push([BP]); (* Word push *)
```

```
                    FI;
                FI;
    OD;
FI;

IF 64-Bit Mode (StackSize = 64)
    THEN
        Push(FrameTemp); (* Quadword push *)
    ELSE IF OperandSize = 32
        THEN
            Push(FrameTemp); FI; (* Doubleword push *)
    ELSE (* OperandSize = 16 *)
            Push(FrameTemp); (* Word push *)
FI;

CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
            RBP ←FrameTemp;
            RSP ←RSP –Size;
    ELSE IF StackSize = 32
        THEN
            EBP ←FrameTemp;
            ESP ←ESP –Size; FI;
    ELSE (* StackSize = 16 *)
            BP ←FrameTemp;
            SP ←SP –Size;
FI;

END;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #SS(0) | If the new value of the SP or ESP register is outside the stack segment limit. |
| #PF(fault-code) | If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #SS(0) | If the new value of the SP or ESP register is outside the stack segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #SS(0) | If the new value of the SP or ESP register is outside the stack segment limit. |
| #PF(fault-code) | If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If the stack address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault. |
| #UD | If the LOCK prefix is used. |

# F2XM1—Compute $2^x-1$

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F0 | F2XM1 | Valid | Valid | Replace ST(0) with ($2^{ST(0)} - 1$). |

## Description

Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range –1.0 to +1.0. If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-21. Results Obtained from F2XM1**

| ST(0) SRC | ST(0) DEST |
|---|---|
| –1.0 to –0 | –0.5 to –0 |
| –0 | –0 |
| +0 | +0 |
| +0 to +1.0 | +0 to 1.0 |

Values other than 2 can be exponentiated using the following formula:

$$x^y \leftarrow 2^{(y * \log_2 x)}$$

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

$ST(0) \leftarrow (2^{ST(0)} - 1)$;

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred.<br>Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value or unsupported format. |
| #D | Source is a denormal value. |

| | |
|---|---|
| #U | Result is too small for destination format. |
| #P | Value cannot be represented exactly in destination format. |

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FABS—Absolute Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 E1 | FABS | Valid | Valid | Replace ST with its absolute value. |

## Description

Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

**Table 3-22. Results Obtained from FABS**

| ST(0) SRC | ST(0) DEST |
|---|---|
| -∞ | +∞ |
| -F | +F |
| -0 | +0 |
| -0 | +0 |
| -F | +F |
| +∞ | +∞ |
| NaN | NaN |

**NOTES:**

F Means finite floating-point value.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

ST(0) ← |ST(0)|;

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; otherwise, set to 0. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FADD/FADDP/FIADD—Add

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D8 /0 | FADD *m32fp* | Valid | Valid | Add *m32fp* to ST(0) and store result in ST(0). |
| DC /0 | FADD *m64fp* | Valid | Valid | Add *m64fp* to ST(0) and store result in ST(0). |
| D8 C0+i | FADD ST(0), ST(i) | Valid | Valid | Add ST(0) to ST(i) and store result in ST(0). |
| DC C0+i | FADD ST(i), ST(0) | Valid | Valid | Add ST(i) to ST(0) and store result in ST(i). |
| DE C0+i | FADDP ST(i), ST(0) | Valid | Valid | Add ST(0) to ST(i), store result in ST(i), and pop the register stack. |
| DE C1 | FADDP | Valid | Valid | Add ST(0) to ST(1), store result in ST(1), and pop the register stack. |
| DA /0 | FIADD *m32int* | Valid | Valid | Add *m32int* to ST(0) and store result in ST(0). |
| DE /0 | FIADD *m16int* | Valid | Valid | Add *m16int* to ST(0) and store result in ST(0). |

## Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward −∞ mode, in which case the result is −0. When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-23.

**Table 3-23. FADD/FADDP/FIADD Results**

| | | DEST | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | −∞ | −F | −0 | +0 | +F | +∞ | NaN |
| SRC | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | * | NaN |
| | −F or −I | −∞ | −F | SRC | SRC | ±F or ±0 | +∞ | NaN |
| | −0 | −∞ | DEST | −0 | ±0 | DEST | +∞ | NaN |
| | +0 | −∞ | DEST | ±0 | +0 | DEST | +∞ | NaN |
| | +F or +I | −∞ | ±F or ±0 | SRC | SRC | +F | +∞ | NaN |
| | +∞ | * | +∞ | +∞ | +∞ | +∞ | +∞ | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF Instruction = FIADD
    THEN
        DEST ← DEST + ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST ← DEST + SRC;
FI;

IF Instruction = FADDP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Operand is an SNaN value or unsupported format. |
| | Operands are infinities of unlike sign. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #O | Result is too large for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FBLD—Load Binary Coded Decimal

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DF /4 | FBLD *m80 dec* | Valid | Valid | Convert BCD value to floating-point and push onto the FPU stack. |

## Description

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of –0.

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
TOP ←TOP –1;
ST(0) ←ConvertToDoubleExtendedPrecisionFP(SRC);
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 1 if stack overflow occurred; otherwise, set to 0. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack overflow occurred. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FBSTP—Store BCD Integer and Pop

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DF /6 | FBSTP m80bcd | Valid | Valid | Store ST(0) in m80bcd and pop ST(0). |

## Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

**Table 3-24. FBSTP Results**

| ST(0) | DEST |
|---|---|
| -∞or Value Too Large for DEST Format | * |
| F ≤-1 | -D |
| -1 < F < -0 | ** |
| -0 | -0 |
| +0 | +0 |
| +0 < F < +1 | ** |
| F ≥ +1 | +D |
| +∞or Value Too Large for DEST Format | * |
| NaN | * |

**NOTES:**

F Means finite floating-point value.

D Means packed-BCD number.

* Indicates floating-point invalid-operation (#IA) exception.

** ±0 or ±1, depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the desti-

nation operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
DEST ←BCD(ST(0));
PopRegisterStack;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Converted value that exceeds 18 BCD digits in length. |
| | Source operand is an SNaN, QNaN, $\pm\infty$, or in an unsupported format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a segment register is being loaded with a segment selector that points to a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FCHS—Change Sign

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 E0 | FCHS | Valid | Valid | Complements sign of ST(0). |

## Description

Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa. The following table shows the results obtained when changing the sign of various classes of numbers.

**Table 3-25. FCHS Results**

| ST(0) SRC | ST(0) DEST |
|---|---|
| -∞ | +∞ |
| -F | +F |
| -0 | +0 |
| +0 | -0 |
| +F | -F |
| +∞ | -∞ |
| NaN | NaN |

**NOTES:**

* F means finite floating-point value.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
SignBit(ST(0)) ←NOT (SignBit(ST(0)));
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; otherwise, set to 0. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FCLEX/FNCLEX—Clear Exceptions

| Opcode* | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 9B DB E2 | FCLEX | Valid | Valid | Clear floating-point exception flags after checking for pending unmasked floating-point exceptions. |
| DB E2 | FNCLEX* | Valid | Valid | Clear floating-point exception flags without checking for pending unmasked floating-point exceptions. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

## IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS* compatibility mode, it is possible (under unusual circumstances) for an FNCLEX instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNCLEX instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCRS register.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
FPUStatusWord[0:7] ←0;
FPUStatusWord[15] ←0;
```

### FPU Flags Affected

The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FCMOV*cc*—Floating-Point Conditional Move

| Opcode* | Instruction | 64-Bit Mode | Compat/ Leg Mode* | Description |
|---|---|---|---|---|
| DA C0+i | FCMOVB ST(0), ST(i) | Valid | Valid | Move if below (CF=1). |
| DA C8+i | FCMOVE ST(0), ST(i) | Valid | Valid | Move if equal (ZF=1). |
| DA D0+i | FCMOVBE ST(0), ST(i) | Valid | Valid | Move if below or equal (CF=1 or ZF=1). |
| DA D8+i | FCMOVU ST(0), ST(i) | Valid | Valid | Move if unordered (PF=1). |
| DB C0+i | FCMOVNB ST(0), ST(i) | Valid | Valid | Move if not below (CF=0). |
| DB C8+i | FCMOVNE ST(0), ST(i) | Valid | Valid | Move if not equal (ZF=0). |
| DB D0+i | FCMOVNBE ST(0), ST(i) | Valid | Valid | Move if not below or equal (CF=0 and ZF=0). |
| DB D8+i | FCMOVNU ST(0), ST(i) | Valid | Valid | Move if not unordered (PF=0). |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

### Description

Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The condition for each mnemonic os given in the Description column above and in Chapter 7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. The source operand is always in the ST(i) register and the destination operand is always ST(0).

The FCMOV*cc* instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor may not support the FCMOV*cc* instructions. Software can check if the FCMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS" in this chapter). If both the CMOV and FPU feature bits are set, the FCMOV*cc* instructions are supported.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

The FCMOVcc instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

## Operation

```
IF condition TRUE
    THEN ST(0) ←ST(i);
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |

## Integer Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FCOM/FCOMP/FCOMPP—Compare Floating Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D8 /2 | FCOM *m32fp* | Valid | Valid | Compare ST(0) with *m32fp*. |
| DC /2 | FCOM *m64fp* | Valid | Valid | Compare ST(0) with *m64fp*. |
| D8 D0+i | FCOM ST(i) | Valid | Valid | Compare ST(0) with ST(i). |
| D8 D1 | FCOM | Valid | Valid | Compare ST(0) with ST(1). |
| D8 /3 | FCOMP *m32fp* | Valid | Valid | Compare ST(0) with *m32fp* and pop register stack. |
| DC /3 | FCOMP *m64fp* | Valid | Valid | Compare ST(0) with m64fp and pop register stack. |
| D8 D8+i | FCOMP ST(i) | Valid | Valid | Compare ST(0) with ST(i) and pop register stack. |
| D8 D9 | FCOMP | Valid | Valid | Compare ST(0) with ST(1) and pop register stack. |
| DE D9 | FCOMPP | Valid | Valid | Compare ST(0) with ST(1) and pop register stack twice. |

## Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that –0.0 is equal to +0.0.

**Table 3-26. FCOM/FCOMP/FCOMPP Results**

| Condition | C3 | C2 | C0 |
|---|---|---|---|
| ST(0) > SRC | 0 | 0 | 0 |
| ST(0) < SRC | 0 | 0 | 1 |
| ST(0) = SRC | 1 | 0 | 0 |
| Unordered* | 1 | 1 | 1 |

**NOTES:**

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared (see "FXAM—Examine ModR/M" in this chapter). If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to "unordered." If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
CASE (relation of operands) OF
    ST > SRC:       C3, C2, C0 ←000;
    ST < SRC:       C3, C2, C0 ←001;
    ST = SRC:       C3, C2, C0 ←100;
ESAC;

IF ST(0) or SRC = NaN or unsupported format
    THEN
        #IA
        IF FPUControlWord.IM = 1
            THEN
                C3, C2, C0 ←111;
        FI;
FI;

IF Instruction = FCOMP
    THEN
        PopRegisterStack;
FI;

IF Instruction = FCOMPP
    THEN
        PopRegisterStack;
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; otherwise, set to 0. |
| C0, C2, C3 | See table on previous page. |

### Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | One or both operands are NaN values or have unsupported formats. |
| | Register is marked empty. |
| #D | One or both operands are denormal values. |

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DB F0+i | FCOMI ST, ST(i) | Valid | Valid | Compare ST(0) with ST(i) and set status flags accordingly. |
| DF F0+i | FCOMIP ST, ST(i) | Valid | Valid | Compare ST(0) with ST(i), set status flags accordingly, and pop register stack. |
| DB E8+i | FUCOMI ST, ST(i) | Valid | Valid | Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly. |
| DF E8+i | FUCOMIP ST, ST(i) | Valid | Valid | Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack. |

## Description

Performs an unordered comparison of the contents of registers ST(0) and ST(i) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that –0.0 is equal to +0.0.

**Table 3-27. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results**

| Comparison Results* | ZF | PF | CF |
|---|---|---|---|
| ST0 > ST(i) | 0 | 0 | 0 |
| ST0 < ST(i) | 0 | 0 | 1 |
| ST0 = ST(i) | 1 | 0 | 0 |
| Unordered** | 1 | 1 | 1 |

**NOTES:**

* See the IA-32 Architecture Compatibility section below.

** Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see "FXAM—ExamineModR/M" in this chapter). The FUCOMI/FUCOMIP instructions perform the same operations as the FCOMI/FCOMIP instructions. The only difference is that the FUCOMI/FUCOMIP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOMI/FCOMIP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

If the operation results in an invalid-arithmetic-operand exception being raised, the status flags in the EFLAGS register are set only if the exception is masked.

The FCOMI/FCOMIP and FUCOMI/FUCOMIP instructions clear the OF flag in the EFLAGS register (regardless of whether an invalid-operation exception is detected).

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

## Operation

```
CASE (relation of operands) OF
    ST(0) > ST(i):      ZF, PF, CF ←000;
    ST(0) < ST(i):      ZF, PF, CF ←001;
    ST(0) = ST(i):      ZF, PF, CF ←100;
ESAC;

IF Instruction is FCOMI or FCOMIP
    THEN
        IF ST(0) or ST(i) = NaN or unsupported format
            THEN
                #IA
                IF FPUControlWord.IM = 1
                    THEN
                        ZF, PF, CF ←111;
                FI;
        FI;
FI;

IF Instruction is FUCOMI or FUCOMIP
    THEN
        IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format
            THEN
                ZF, PF, CF ←111;
            ELSE (* ST(0) or ST(i) is SNaN or unsupported format *)
                 #IA;
                IF FPUControlWord.IM = 1
                    THEN
```

```
                    ZF, PF, CF ←111;
                FI;
        FI;
FI;

IF Instruction is FCOMIP or FUCOMIP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; otherwise, set to 0. |
| C0, C2, C3 | Not affected. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | (FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats.<br>(FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FCOS—Cosine

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 FF | FCOS | Valid | Valid | Replace ST(0) with its cosine. |

## Description

Computes the cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range $-2^{63}$ to $+2^{63}$. The following table shows the results obtained when taking the cosine of various classes of numbers.

**Table 3-28. FCOS Results**

| ST(0) SRC | ST(0) DEST |
|---|---|
| $-\infty$ | * |
| $-F$ | $-1$ to $+1$ |
| $-0$ | $+1$ |
| $+0$ | $+1$ |
| $+F$ | $-1$ to $+1$ |
| $+\infty$ | * |
| NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range $-2^{63}$ to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of $2\pi$ or by using the FPREM instruction with a divisor of $2\pi$. See the section titled "Pi" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for $\pi$ in performing such reductions.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF |ST(0)| < 2^63
THEN
    C2 ← 0;
```

```
    ST(0) ←cosine(ST(0));
ELSE (* Source operand is out-of-range *)
    C2 ←1;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred.<br>Set if result was rounded up; cleared otherwise.<br>Undefined if C2 is 1. |
| C2 | Set to 1 if outside range ($-2^{63}$ < source operand < $+2^{63}$); otherwise, set to 0. |
| C0, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value, ∞, or unsupported format. |
| #D | Source is a denormal value. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FDECSTP—Decrement Stack-Top Pointer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F6 | FDECSTP | Valid | Valid | Decrement TOP field in FPU status word. |

## Description

Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). If the TOP field contains a 0, it is set to 7. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF TOP = 0
    THEN TOP ←7;
    ELSE TOP ←TOP – 1;
FI;
```

## FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FDIV/FDIVP/FIDIV—Divide

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D8 /6 | FDIV *m32fp* | Valid | Valid | Divide ST(0) by *m32fp* and store result in ST(0). |
| DC /6 | FDIV *m64fp* | Valid | Valid | Divide ST(0) by *m64fp* and store result in ST(0). |
| D8 F0+i | FDIV ST(0), ST(i) | Valid | Valid | Divide ST(0) by ST(i) and store result in ST(0). |
| DC F8+i | FDIV ST(i), ST(0) | Valid | Valid | Divide ST(i) by ST(0) and store result in ST(i). |
| DE F8+i | FDIVP ST(i), ST(0) | Valid | Valid | Divide ST(i) by ST(0), store result in ST(i), and pop the register stack. |
| DE F9 | FDIVP | Valid | Valid | Divide ST(1) by ST(0), store result in ST(1), and pop the register stack. |
| DA /6 | FIDIV *m32int* | Valid | Valid | Divide ST(0) by *m32int* and store result in ST(0). |
| DE /6 | FIDIV *m16int* | Valid | Valid | Divide ST(0) by *m64int* and store result in ST(0). |

## Description

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(i) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to double extended-precision floating-point format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-29. FDIV/FDIVP/FIDIV Results**

| | | DEST | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | -∞ | -F | -0 | +0 | +F | +∞ | NaN |
| SRC | -∞ | * | -0 | -0 | +0 | +0 | * | NaN |
| | -F | +∞ | -F | -0 | +0 | +F | -∞ | NaN |
| | -I | +∞ | -F | -0 | +0 | +F | -∞ | NaN |
| | -0 | +∞ | ** | * | * | ** | -∞ | NaN |
| | +0 | -∞ | ** | * | * | ** | +∞ | NaN |
| | +I | -∞ | -F | -0 | +0 | +F | +∞ | NaN |
| | +F | -∞ | -F | -0 | +0 | +F | +∞ | NaN |
| | +∞ | * | -0 | -0 | +0 | +0 | * | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF SRC = 0
    THEN
        #Z;
    ELSE
        IF Instruction is FIDIV
            THEN
                DEST ←DEST / ConvertToDoubleExtendedPrecisionFP(SRC);
            ELSE (* Source operand is floating-point value *)
                DEST ←DEST / SRC;
        FI;
FI;
```

```
IF Instruction = FDIVP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Operand is an SNaN value or unsupported format. |
| | ±∞ / ±∞; ±0 / ±0 |
| #D | Source is a denormal value. |
| #Z | DEST / ±0, where DEST is not equal to ±0. |
| #U | Result is too small for destination format. |
| #O | Result is too large for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FDIVR/FDIVRP/FIDIVR—Reverse Divide

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D8 /7 | FDIVR *m32fp* | Valid | Valid | Divide *m32fp* by ST(0) and store result in ST(0). |
| DC /7 | FDIVR *m64fp* | Valid | Valid | Divide *m64fp* by ST(0) and store result in ST(0). |
| D8 F8+i | FDIVR ST(0), ST(i) | Valid | Valid | Divide ST(i) by ST(0) and store result in ST(0). |
| DC F0+i | FDIVR ST(i), ST(0) | Valid | Valid | Divide ST(0) by ST(i) and store result in ST(i). |
| DE F0+i | FDIVRP ST(i), ST(0) | Valid | Valid | Divide ST(0) by ST(i), store result in ST(i), and pop the register stack. |
| DE F1 | FDIVRP | Valid | Valid | Divide ST(0) by ST(1), store result in ST(1), and pop the register stack. |
| DA /7 | FIDIVR *m32int* | Valid | Valid | Divide *m32int* by ST(0) and store result in ST(0). |
| DE /7 | FIDIVR *m16int* | Valid | Valid | Divide *m16int* by ST(0) and store result in ST(0). |

## Description

Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a floating-point or an integer value) by the contents of the ST(0) register. The two-operand version, divides the contents of the ST(i) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to double extended-precision floating-point format before performing the division.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-30. FDIVR/FDIVRP/FIDIVR Results**

| | | DEST | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | -∞ | -F | -0 | +0 | +F | +∞ | NaN |
| SRC | -∞ | * | +∞ | +∞ | -∞ | -∞ | * | NaN |
| | -F | +0 | +F | ** | ** | -F | -0 | NaN |
| | -I | +0 | +F | ** | ** | -F | -0 | NaN |
| | -0 | +0 | +0 | * | * | -0 | -0 | NaN |
| | +0 | -0 | -0 | * | * | +0 | +0 | NaN |
| | +I | -0 | -F | ** | ** | +F | +0 | NaN |
| | +F | -0 | -F | ** | ** | +F | +0 | NaN |
| | +∞ | * | -∞ | -∞ | +∞ | +∞ | * | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the source operand is an integer 0, it is treated as a +0. This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF DEST = 0
    THEN
        #Z;
    ELSE
        IF Instruction = FIDIVR
            THEN
                DEST ←ConvertToDoubleExtendedPrecisionFP(SRC) / DEST;
            ELSE (* Source operand is floating-point value *)
```

```
                DEST ←SRC / DEST;
        FI;
FI;

IF Instruction = FDIVRP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Operand is an SNaN value or unsupported format. |
| | ±∞ / ±∞; ±0 / ±0 |
| #D | Source is a denormal value. |
| #Z | SRC / ±0, where SRC is not equal to ±0. |
| #U | Result is too small for destination format. |
| #O | Result is too large for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FFREE—Free Floating-Point Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DD C0+i | FFREE ST(i) | Valid | Valid | Sets tag for ST(i) to empty. |

## Description

Sets the tag in the FPU tag register associated with register ST(i) to empty (11B). The contents of ST(i) and the FPU stack-top pointer (TOP) are not affected.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

TAG(i) ←11B;

## FPU Flags Affected

C0, C1, C2, C3 undefined.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FICOM/FICOMP—Compare Integer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DE /2 | FICOM *m16int* | Valid | Valid | Compare ST(0) with *m16int.* |
| DA /2 | FICOM *m32int* | Valid | Valid | Compare ST(0) with *m32int.* |
| DE /3 | FICOMP *m16int* | Valid | Valid | Compare ST(0) with *m16int* and pop stack register. |
| DA /3 | FICOMP *m32int* | Valid | Valid | Compare ST(0) with *m32int* and pop stack register. |

## Description

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to double extended-precision floating-point format before the comparison is made.

**Table 3-31. FICOM/FICOMP Results**

| Condition | C3 | C2 | C0 |
|---|---|---|---|
| ST(0) > SRC | 0 | 0 | 0 |
| ST(0) < SRC | 0 | 0 | 1 |
| ST(0) = SRC | 1 | 0 | 0 |
| Unordered | 1 | 1 | 1 |

These instructions perform an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—ExamineModR/M" in this chapter). If either operand is a NaN or is in an undefined format, the condition flags are set to "unordered."

The sign of zero is ignored, so that –0.0 ←→ 0.0.

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
CASE (relation of operands) OF
    ST(0) > SRC:    C3, C2, C0 ← 000;
    ST(0) < SRC:    C3, C2, C0 ← 001;
    ST(0) = SRC:    C3, C2, C0 ← 100;
    Unordered:      C3, C2, C0 ← 111;
```

```
ESAC;

IF Instruction = FICOMP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; otherwise, set to 0. |
| C0, C2, C3 | See table on previous page. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | One or both operands are NaN values or have unsupported formats. |
| #D | One or both operands are denormal values. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FILD—Load Integer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DF /0 | FILD *m16int* | Valid | Valid | Push *m16int* onto the FPU register stack. |
| DB /0 | FILD *m32int* | Valid | Valid | Push *m32int* onto the FPU register stack. |
| DF /5 | FILD *m64int* | Valid | Valid | Push *m64int* onto the FPU register stack. |

## Description

Converts the signed-integer source operand into double extended-precision floating-point format and pushes the value onto the FPU register stack. The source operand can be a word, doubleword, or quadword integer. It is loaded without rounding errors. The sign of the source operand is preserved.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
TOP ←TOP –1;
ST(0) ←ConvertToDoubleExtendedPrecisionFP(SRC);
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 1 if stack overflow occurred; set to 0 otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack overflow occurred. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FINCSTP—Increment Stack-Top Pointer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F7 | FINCSTP | Valid | Valid | Increment the TOP field in the FPU status register. |

## Description

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). If the TOP field contains a 7, it is set to 0. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF TOP = 7
    THEN TOP ←0;
    ELSE TOP ←TOP +1;
FI;
```

## FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FINIT/FNINIT—Initialize Floating-Point Unit

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 9B DB E3 | FINIT | Valid | Valid | Initialize FPU after checking for pending unmasked floating-point exceptions. |
| DB E3 | FNINIT* | Valid | Valid | Initialize FPU without checking for pending unmasked floating-point exceptions. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

The assembler issues two instructions for the FINIT instruction (an FWAIT instruction followed by an FNINIT instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNINIT instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNINIT instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

In the Intel387 math coprocessor, the FINIT/FNINIT instruction does not clear the instruction and data pointers.

This instruction affects only the x87 FPU. It does not affect the XMM and MXCSR registers.

## Operation

```
FPUControlWord ←037FH;
FPUStatusWord ←0;
FPUTagWord ←FFFFH;
FPUDataPointer ←0;
FPUInstructionPointer ←0;
FPULastInstructionOpcode ←0;
```

## FPU Flags Affected

C0, C1, C2, C3 set to 0.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FIST/FISTP—Store Integer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DF /2 | FIST *m16int* | Valid | Valid | Store ST(0) in *m16int.* |
| DB /2 | FIST *m32int* | Valid | Valid | Store ST(0) in *m32int.* |
| DF /3 | FISTP *m16int* | Valid | Valid | Store ST(0) in *m16int* and pop register stack. |
| DB /3 | FISTP *m32int* | Valid | Valid | Store ST(0) in *m32int* and pop register stack. |
| DF /7 | FISTP *m64int* | Valid | Valid | Store ST(0) in *m64int* and pop register stack. |

### Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word or double-word integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction also stores values in quadword integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

**Table 3-32. FIST/FISTP Results**

| ST(0) | DEST |
|---|---|
| −∞ or Value Too Large for DEST Format | * |
| F ≤ −1 | - I |
| −1 < F < −0 | ** |
| −0 | 0 |
| +0 | 0 |
| +0 < F < +1 | ** |
| F ≥ +1 | + I |
| +∞ or Value Too Large for DEST Format | * |

**Table 3-32. FIST/FISTP Results (Contd.)**

| ST(0) | DEST |
|---|---|
| NaN | * |

**NOTES:**
F Means finite floating-point value.
I Means integer.
* Indicates floating-point invalid-operation (#IA) exception.
** 0 or ±1, depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in memory.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
DEST ←Integer(ST(0));

IF Instruction = FISTP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred.<br>Indicates rounding direction of if the inexact exception (#P) is generated: 0 ←not roundup; 1 ←roundup.<br>Set to 0 otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Converted value is too large for the destination format.<br>Source operand is an SNaN, QNaN, ±∞, or unsupported format. |
| #P | Value cannot be represented exactly in destination format. |

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FISTTP—Store Integer with Truncation

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DF /1 | FISTTP *m16int* | Valid | Valid | Store ST(0) in *m16int with truncation.* |
| DB /1 | FISTTP *m32int* | Valid | Valid | Store ST(0) in *m32int with truncation.* |
| DD /1 | FISTTP *m64int* | Valid | Valid | Store ST(0) in *m64int* with truncation. |

## Description

FISTTP converts the value in ST into a signed integer using truncation (chop) as rounding mode, transfers the result to the destination, and pop ST. FISTTP accepts word, short integer, and long integer destinations.

The following table shows the results obtained when storing various classes of numbers in integer format.

**Table 3-33. FISTTP Results**

| ST(0) | DEST |
|---|---|
| $-\infty$ or Value Too Large for DEST Format | * |
| $F \leq -1$ | –I |
| $-1 < F < +1$ | 0 |
| $F \geq +1$ | + I |
| $+\infty$ or Value Too Large for DEST Format | * |
| NaN | * |

**NOTES:**

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
DEST ← ST;
pop ST;
```

## Flags Affected

C1 is cleared; C0, C2, C3 undefined.

### Numeric Exceptions

Invalid, Stack Invalid (stack underflow), Precision.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is in a nonwritable segment. |
| | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #NM | If CR0.EM[bit 2] = 1. |
| | If CR0.TS[bit 3] = 1. |
| #UD | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| #NM | If CR0.EM[bit 2] = 1. |
| | If CR0.TS[bit 3] = 1. |
| #UD | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| #NM | If CR0.EM[bit 2] = 1. |
| | If CR0.TS[bit 3] = 1. |
| #UD | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | For a page fault. |
| #AC(0) | For unaligned memory reference if the current privilege is 3. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.<br>If the LOCK prefix is used. |

# FLD—Load Floating Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 /0 | FLD *m32fp* | Valid | Valid | Push *m32fp* onto the FPU register stack. |
| DD /0 | FLD *m64fp* | Valid | Valid | Push *m64fp* onto the FPU register stack. |
| DB /5 | FLD *m80fp* | Valid | Valid | Push *m80fp* onto the FPU register stack. |
| D9 C0+i | FLD ST(i) | Valid | Valid | Push ST(i) onto the FPU register stack. |

## Description

Pushes the source operand onto the FPU register stack. The source operand can be in single-precision, double-precision, or double extended-precision floating-point format. If the source operand is in single-precision or double-precision floating-point format, it is automatically converted to the double extended-precision floating-point format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(i)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF SRC is ST(i)
    THEN
        temp ←ST(i);
FI;

TOP ←TOP –1;

IF SRC is memory-operand
    THEN
        ST(0) ←ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* SRC is ST(i) *)
        ST(0) ←temp;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 1 if stack overflow occurred; otherwise, set to 0. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow or overflow occurred. |

| | |
|---|---|
| #IA | Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD m80fp or FLD ST(i)). |
| #D | Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format. |

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If destination is located in a non-writable segment.<br>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

| Opcode* | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 E8 | FLD1 | Valid | Valid | Push +1.0 onto the FPU register stack. |
| D9 E9 | FLDL2T | Valid | Valid | Push $\log_2 10$ onto the FPU register stack. |
| D9 EA | FLDL2E | Valid | Valid | Push $\log_2 e$ onto the FPU register stack. |
| D9 EB | FLDPI | Valid | Valid | Push $\pi$ onto the FPU register stack. |
| D9 EC | FLDLG2 | Valid | Valid | Push $\log_{10} 2$ onto the FPU register stack. |
| D9 ED | FLDLN2 | Valid | Valid | Push $\log_e 2$ onto the FPU register stack. |
| D9 EE | FLDZ | Valid | Valid | Push +0.0 onto the FPU register stack. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Push one of seven commonly used constants (in double extended-precision floating-point format) onto the FPU register stack. The constants that can be loaded with these instructions include +1.0, +0.0, $\log_2 10$, $\log_2 e$, $\pi$, $\log_{10} 2$, and $\log_e 2$. For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to double extended-precision floating-point format. The inexact-result exception (#P) is not generated as a result of the rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up.

See the section titled “Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the $\pi$ constant.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel 287 math coprocessors.

## Operation

```
TOP ←TOP –1;
ST(0) ←CONSTANT;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 1 if stack overflow occurred; otherwise, set to 0. |
| C0, C2, C3 | Undefined. |

### Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack overflow occurred. |

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FLDCW—Load x87 FPU Control Word

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 /5 | FLDCW m2byte | Valid | Valid | Load FPU control word from *m2byte*. |

## Description

Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU's mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmasks one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
FPUControlWord ←SRC;
```

## FPU Flags Affected

C0, C1, C2, C3 undefined.

## Floating-Point Exceptions

None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next "waiting" floating-point instruction.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FLDENV—Load x87 FPU Environment

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 /4 | FLDENV *m14/28byte* | Valid | Valid | Load FPU environment from *m14byte* or *m28byte.* |

## Description

Loads the complete x87 FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory. This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

If a page or limit fault occurs during the execution of this instruction, the state of the x87 FPU registers as seen by the fault handler may be different than the state being loaded from memory. In such situations, the fault handler should ignore the status of the x87 FPU registers, handle the fault, and return. The FLDENV instruction will then complete the loading of the x87 FPU registers with no resulting context inconsistency.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
FPUControlWord ←SRC[FPUControlWord];
FPUStatusWord ←SRC[FPUStatusWord];
FPUTagWord ←SRC[FPUTagWord];
FPUDataPointer ←SRC[FPUDataPointer];
FPUInstructionPointer ←SRC[FPUInstructionPointer];
FPULastInstructionOpcode ←SRC[FPULastInstructionOpcode];
```

## FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

## Floating-Point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next “waiting” floating-point instruction.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FMUL/FMULP/FIMUL—Multiply

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D8 /1 | FMUL *m32fp* | Valid | Valid | Multiply ST(0) by *m32fp* and store result in ST(0). |
| DC /1 | FMUL *m64fp* | Valid | Valid | Multiply ST(0) by *m64fp* and store result in ST(0). |
| D8 C8+i | FMUL ST(0), ST(i) | Valid | Valid | Multiply ST(0) by ST(i) and store result in ST(0). |
| DC C8+i | FMUL ST(i), ST(0) | Valid | Valid | Multiply ST(i) by ST(0) and store result in ST(i). |
| DE C8+i | FMULP ST(i), ST(0) | Valid | Valid | Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack. |
| DE C9 | FMULP | Valid | Valid | Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack. |
| DA /1 | FIMUL *m32int* | Valid | Valid | Multiply ST(0) by *m32int* and store result in ST(0). |
| DE /1 | FIMUL *m16int* | Valid | Valid | Multiply ST(0) by *m16int* and store result in ST(0). |

## Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be an FPU data register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction multiplies the contents of the ST(1) register by the contents of the ST(0) register and stores the product in the ST(1) register. The one-operand version multiplies the contents of the ST(0) register by the contents of a memory location (either a floating point or an integer value) and stores the product in the ST(0) register. The two-operand version, multiplies the contents of the ST(0) register by the contents of the ST(i) register, or vice versa, with the result being stored in the register specified with the first operand (the destination operand).

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to double extended-precision floating-point format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or ∞. When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-34. FMUL/FMULP/FIMUL Results**

| | | DEST | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | −∞ | −F | −0 | +0 | +F | +∞ | NaN |
| SRC | −∞ | +∞ | +∞ | * | * | −∞ | −∞ | NaN |
| | −F | +∞ | +F | −0 | +0 | −F | −∞ | NaN |
| | −I | +∞ | +F | −0 | +0 | −F | −∞ | NaN |
| | −0 | * | −0 | −0 | +0 | −0 | * | NaN |
| | +0 | * | −0 | −0 | +0 | −0 | * | NaN |
| | +I | −∞ | −F | −0 | +0 | −F | +∞ | NaN |
| | +F | −∞ | −F | −0 | +0 | −F | +∞ | NaN |
| | +∞ | −∞ | −∞ | * | * | +∞ | +∞ | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

I Means Integer.

* Indicates invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF Instruction = FIMUL
    THEN
        DEST ←DEST * ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST ←DEST * SRC;
FI;

IF Instruction = FMULP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Operand is an SNaN value or unsupported format. |
| | One operand is ±0 and the other is ±∞. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #O | Result is too large for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FNOP—No Operation

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 D0 | FNOP | Valid | Valid | No operation is performed. |

## Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## FPU Flags Affected

C0, C1, C2, C3 undefined.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FPATAN—Partial Arctangent

| Opcode* | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F3 | FPATAN | Valid | Valid | Replace ST(1) with arctan(ST(1)/ ST(0)) and pop the register stack. |

NOTES:

* See IA-32 Architecture Compatibility section below.

## Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than $\pi$.

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point ($-X$,Y) is in the second quadrant, resulting in an angle between $\pi/2$ and $\pi$, while a point (X,$-Y$) is in the fourth quadrant, resulting in an angle between 0 and $-\pi/2$. A point ($-X$,$-Y$) is in the third quadrant, giving an angle between $-\pi/2$ and $-\pi$.

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

### Table 3-35. FPATAN Results

| | | ST(0) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | -∞ | -F | -0 | +0 | +F | +∞ | NaN |
| ST(1) | -∞ | -3π/ 4* | -π/2 | -π/2 | -π/2 | -π/2 | -π/4* | NaN |
| | -F | -p | -π to -π/ 2 | -π/2 | -π/2 | -π/ 2 to -0 | -0 | NaN |
| | -0 | -p | -p | -p* | -0* | -0 | -0 | NaN |
| | +0 | +p | +p | +π* | +0* | +0 | +0 | NaN |
| | +F | +p | +π to +π/ 2 | +π/ 2 | +π/ 2 | +π/ 2 to +0 | +0 | NaN |
| | +∞ | +3π/ 4* | +π/ 2 | +π/ 2 | +π/ 2 | +π/ 2 | +π/4* | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Table 8-10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, specifies that the ratios 0/0 and ∞/∞ generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the floating-point QNaN indefinite value is returned. With the FPATAN instruction, the 0/0 or ∞/∞ value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation, arctangent(0,0) etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow floating-point values as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$0 \leq |ST(1)| < |ST(0)| < +\infty$

## Operation

```
ST(1) ← arctan(ST(1) / ST(0));
PopRegisterStack;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

### Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value or unsupported format. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #P | Value cannot be represented exactly in destination format. |

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FPREM—Partial Remainder

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F8 | FPREM | Valid | Valid | Replace ST(0) with the remainder obtained from dividing ST(0) by ST(1). |

## Description

Computes the remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

Remainder ←ST(0) –(Q ∗ ST(1))

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of [ST(0) / ST(1)] toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the inexact-result exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

**Table 3-36. FPREM Results**

| | | ST(1) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | -∞ | -F | -0 | +0 | +F | +∞ | NaN |
| ST(0) | -∞ | * | * | * | * | * | * | NaN |
| | -F | ST(0) | -F or -0 | ** | ** | -F or -0 | ST(0) | NaN |
| | -0 | -0 | -0 | * | * | -0 | -0 | NaN |
| | +0 | +0 | +0 | * | * | +0 | +0 | NaN |
| | +F | ST(0) | -F or -0 | ** | ** | -F or -0 | ST(0) | NaN |
| | +∞ | * | * | * | * | * | * | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name “partial remainder” because of the way it computes the remainder. This instruction arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU status word. This information is important in argument reduction for the tangent function (using a modulus of $\pi/4$), because it locates the original angle in the correct one of eight sectors of the unit circle.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
D ←exponent(ST(0)) – exponent(ST(1));

IF D < 64
    THEN
        Q ←Integer(TruncateTowardZero(ST(0) / ST(1)));
        ST(0) ←ST(0) – (ST(1) * Q);
        C2 ←0;
        C0, C3, C1 ←LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 ←1;
        N ←An implementation-dependent number between 32 and 63;
        QQ ←Integer(TruncateTowardZero((ST(0) / ST(1)) / 2^(D –N)));
        ST(0) ←ST(0) – (ST(1) * QQ * 2^(D –N));
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C0 | Set to bit 2 (Q2) of the quotient. |
| C1 | Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0). |
| C2 | Set to 0 if reduction complete; set to 1 if incomplete. |
| C3 | Set to bit 1 (Q1) of the quotient. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value, modulus is 0, dividend is ∞, or unsupported format. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FPREM1—Partial Remainder

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F5 | FPREM1 | Valid | Valid | Replace ST(0) with the IEEE remainder obtained from dividing ST(0) by ST(1). |

## Description

Computes the IEEE remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

Remainder ←ST(0) –(Q ∗ ST(1))

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of [ST(0) / ST(1)] toward the nearest integer value. The magnitude of the remainder is less than or equal to half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

**Table 3-37. FPREM1 Results**

| | | ST(1) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | -∞ | -F | -0 | +0 | +F | +∞ | NaN |
| ST(0) | -∞ | * | * | * | * | * | * | NaN |
| | -F | ST(0) | ±F or -0 | ** | ** | ±F or -0 | ST(0) | NaN |
| | -0 | -0 | -0 | * | * | -0 | -0 | NaN |
| | +0 | +0 | +0 | * | * | +0 | +0 | NaN |
| | +F | ST(0) | ±F or +0 | ** | ** | ±F or +0 | ST(0) | NaN |
| | +∞ | * | * | * | * | * | * | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Standard 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the "Operation" section below).

Like the FPREM instruction, FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU status word. This information is important in argument reduction for the tangent function (using a modulus of $\pi/4$), because it locates the original angle in the correct one of eight sectors of the unit circle.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
D ←exponent(ST(0)) – exponent(ST(1));

IF D < 64
    THEN
        Q ←Integer(RoundTowardNearestInteger(ST(0) / ST(1)));
        ST(0) ←ST(0) – (ST(1) * Q);
        C2 ←0;
        C0, C3, C1 ←LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 ←1;
        N ←An implementation-dependent number between 32 and 63;
        QQ ←Integer(TruncateTowardZero((ST(0) / ST(1)) / 2^(D –N)));
        ST(0) ←ST(0) – (ST(1) * QQ * 2^(D –N));
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C0 | Set to bit 2 (Q2) of the quotient. |
| C1 | Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0). |

| | |
|---|---|
| C2 | Set to 0 if reduction complete; set to 1 if incomplete. |
| C3 | Set to bit 1 (Q1) of the quotient. |

### Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value, modulus (divisor) is 0, dividend is ∞, or unsupported format. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FPTAN—Partial Tangent

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F2 | FPTAN | Valid | Valid | Replace ST(0) with its tangent and push 1 onto the FPU stack. |

## Description

Computes the tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than $\pm 2^{63}$. The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

**Table 3-38. FPTAN Results**

| ST(0) SRC | ST(0) DEST |
|---|---|
| -∞ | * |
| -F | -F to -F |
| -0 | -0 |
| -0 | -0 |
| -F | -F to -F |
| +∞ | * |
| NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range $-2^{63}$ to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of $2\pi$ or by using the FPREM instruction with a divisor of $2\pi$. See the section titled “Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for $\pi$ in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF ST(0) < 2^63
    THEN
        C2 ←0;
        ST(0) ←tan(ST(0));
        TOP ←TOP –1;
        ST(0) ←1.0;
    ELSE (* Source operand is out-of-range *)
        C2 ←1;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C2 | Set to 1 if outside range ($-2^{63}$ < source operand < $+2^{63}$); otherwise, set to 0. |
| C0, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow or overflow occurred. |
| #IA | Source operand is an SNaN value, ∞, or unsupported format. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FRNDINT—Round to Integer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 FC | FRNDINT | Valid | Valid | Round ST(0) to an integer. |

## Description

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is ∞, the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

ST(0) ←RoundToIntegralValue(ST(0));

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value or unsupported format. |
| #D | Source operand is a denormal value. |
| #P | Source operand is not an integral value. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FRSTOR—Restore x87 FPU State

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DD /4 | FRSTOR *m94/108byte* | Valid | Valid | Load FPU state from *m94byte* or *m108byte*. |

## Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated. To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
FPUControlWord ←SRC[FPUControlWord];
FPUStatusWord ←SRC[FPUStatusWord];
FPUTagWord ←SRC[FPUTagWord];
FPUDataPointer ←SRC[FPUDataPointer];
FPUInstructionPointer ←SRC[FPUInstructionPointer];
FPULastInstructionOpcode ←SRC[FPULastInstructionOpcode];

ST(0) ←SRC[ST(0)];
ST(1) ←SRC[ST(1)];
ST(2) ←SRC[ST(2)];
ST(3) ←SRC[ST(3)];
ST(4) ←SRC[ST(4)];
ST(5) ←SRC[ST(5)];
ST(6) ←SRC[ST(6)];
```

ST(7) ←SRC[ST(7)];

## FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

## Floating-Point Exceptions

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FSAVE/FNSAVE—Store x87 FPU State

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 9B DD /6 | FSAVE *m94/108byte* | Valid | Valid | Store FPU state to *m94byte* or *m108byte* after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU. |
| DD /6 | FNSAVE* *m94/108byte* | Valid | Valid | Store FPU environment to *m94byte* or *m108byte* without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see "FINIT/FNINIT—Initialize Floating-Point Unit" in this chapter).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a "clean" FPU to a procedure.

The assembler issues two instructions for the FSAVE instruction (an FWAIT instruction followed by an FNSAVE instruction), and the processor executes each of these

instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

For Intel math coprocessors and FPUs prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps insure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

## Operation

```
(* Save FPU State and Registers *)

DEST[FPUControlWord] ←FPUControlWord;
DEST[FPUStatusWord] ←FPUStatusWord;
DEST[FPUTagWord] ←FPUTagWord;
DEST[FPUDataPointer] ←FPUDataPointer;
DEST[FPUInstructionPointer] ←FPUInstructionPointer;
DEST[FPULastInstructionOpcode] ←FPULastInstructionOpcode;

DEST[ST(0)] ←ST(0);
DEST[ST(1)] ←ST(1);
DEST[ST(2)] ←ST(2);
DEST[ST(3)] ←ST(3);
DEST[ST(4)]←ST(4);
DEST[ST(5)] ←ST(5);
DEST[ST(6)] ←ST(6);
DEST[ST(7)] ←ST(7);

(* Initialize FPU *)

FPUControlWord ←037FH;
FPUStatusWord ←0;
FPUTagWord ←FFFFH;
FPUDataPointer ←0;
FPUInstructionPointer ←0;
FPULastInstructionOpcode ←0;
```

## FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# FSCALE—Scale

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 FD | FSCALE | Valid | Valid | Scale ST(0) by ST(1). |

## Description

Truncates the value in the source operand (toward 0) to an integral value and adds that value to the exponent of the destination operand. The destination and source operands are floating-point values located in registers ST(0) and ST(1), respectively. This instruction provides rapid multiplication or division by integral powers of 2. The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-39. FSCALE Results**

| | ST(1) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | -∞ | -F | -0 | +0 | +F | +∞ | NaN |
| ST(0) | -∞ | NaN | -∞ | -∞ | -∞ | -∞ | -∞ | NaN |
| | -F | -0 | -F | -F | -F | -F | -∞ | NaN |
| | -0 | -0 | -0 | -0 | -0 | -0 | NaN | NaN |
| | +0 | -0 | -0 | -0 | +0 | +0 | NaN | NaN |
| | +F | -0 | +F | +F | +F | +F | +∞ | NaN |
| | +∞ | NaN | +∞ | +∞ | +∞ | +∞ | +∞ | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value

before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

$ST(0) \leftarrow ST(0) * 2^{RoundTowardZero(ST(1))};$

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value or unsupported format. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #O | Result is too large for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FSIN—Sine

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 FE | FSIN | Valid | Valid | Replace ST(0) with its sine. |

## Description

Computes the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range $-2^{63}$ to $+2^{63}$. The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

**Table 3-40. FSIN Results**

| SRC (ST(0)) | DEST (ST(0)) |
|---|---|
| −∞ | * |
| −F | −1 to +1 |
| −0 | −0 |
| +0 | +0 |
| +F | −1 to +1 |
| +∞ | * |
| NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range $-2^{63}$ to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of $2\pi$ or by using the FPREM instruction with a divisor of $2\pi$. See the section titled “Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for $\pi$ in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF ST(0) < 2^63
    THEN
        C2 ← 0;
```

```
        ST(0) ←sin(ST(0));
    ELSE (* Source operand out of range *)
        C2 ←1;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred.<br>Set if result was rounded up; cleared otherwise. |
| C2 | Set to 1 if outside range ($-2^{63}$ < source operand < $+2^{63}$); otherwise, set to 0. |
| C0, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value, ∞, or unsupported format. |
| #D | Source operand is a denormal value. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FSINCOS—Sine and Cosine

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 FB | FSINCOS | Valid | Valid | Compute the sine and cosine of ST(0); replace ST(0) with the sine, and push the cosine onto the register stack. |

## Description

Computes both the sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range $-2^{63}$ to $+2^{63}$. The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

**Table 3-41. FSINCOS Results**

| SRC | DEST | |
|---|---|---|
| ST(0) | ST(1) Cosine | ST(0) Sine |
| $-\infty$ | * | * |
| −F | −1 to +1 | −1 to +1 |
| −0 | +1 | −0 |
| +0 | +1 | +0 |
| +F | −1 to +1 | −1 to +1 |
| $+\infty$ | * | * |
| NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range $-2^{63}$ to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of $2\pi$ or by using the FPREM instruction with a divisor of $2\pi$. See the section titled “Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for $\pi$ in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF ST(0) < 2^63
    THEN
        C2 ←0;
        TEMP ←cosine(ST(0));
        ST(0) ←sine(ST(0));
        TOP ←TOP –1;
        ST(0) ←TEMP;
    ELSE (* Source operand out of range *)
        C2 ←1;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; set to 1 of stack overflow occurs.<br>Set if result was rounded up; cleared otherwise. |
| C2 | Set to 1 if outside range ($-2^{63}$ < source operand < $+2^{63}$); otherwise, set to 0. |
| C0, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow or overflow occurred. |
| #IA | Source operand is an SNaN value, ∞, or unsupported format. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FSQRT—Square Root

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 FA | FSQRT | Valid | Valid | Computes square root of ST(0) and stores the result in ST(0). |

## Description

Computes the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-42. FSQRT Results**

| SRC (ST(0)) | DEST (ST(0)) |
|---|---|
| -∞ | * |
| -F | * |
| -0 | -0 |
| -0 | -0 |
| -F | -F |
| +∞ | +∞ |
| NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
ST(0) ←SquareRoot(ST(0));
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred.<br>Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value or unsupported format. |

| | |
|---|---|
| | Source operand is a negative value (except for -0). |
| #D | Source operand is a denormal value. |
| #P | Value cannot be represented exactly in destination format. |

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FST/FSTP—Store Floating Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 /2 | FST *m32fp* | Valid | Valid | Copy ST(0) to *m32fp*. |
| DD /2 | FST *m64fp* | Valid | Valid | Copy ST(0) to *m64fp*. |
| DD D0+i | FST ST(i) | Valid | Valid | Copy ST(0) to ST(i). |
| D9 /3 | FSTP *m32fp* | Valid | Valid | Copy ST(0) to *m32fp* and pop register stack. |
| DD /3 | FSTP *m64fp* | Valid | Valid | Copy ST(0) to *m64fp* and pop register stack. |
| DB /7 | FSTP *m80fp* | Valid | Valid | Copy ST(0) to *m80fp* and pop register stack. |
| DD D8+i | FSTP ST(i) | Valid | Valid | Copy ST(0) to ST(i) and pop register stack. |

## Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU register stack. When storing the value in memory, the value is converted to single-precision or double-precision floating-point format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also store values in memory in double extended-precision floating-point format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single-precision or double-precision, the significand of the value being stored is rounded to the width of the destination (according to the rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is ±0, ±∞, or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value's identity as a 0, ∞, or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
DEST ←ST(0);

IF Instruction = FSTP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred.<br>Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 ←not roundup; 1 ←roundup. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Source operand is an SNaN value or unsupported format. Does not occur if the source operand is in double extended-precision floating-point format. |
| #U | Result is too small for the destination format. |
| #O | Result is too large for the destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment.<br>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FSTCW/FNSTCW—Store x87 FPU Control Word

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 9B D9 /7 | FSTCW *m2byte* | Valid | Valid | Store FPU control word to *m2byte* after checking for pending unmasked floating-point exceptions. |
| D9 /7 | FNSTCW* *m2byte* | Valid | Valid | Store FPU control word to *m2byte* without checking for pending unmasked floating-point exceptions. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

The assembler issues two instructions for the FSTCW instruction (an FWAIT instruction followed by an FNSTCW instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTCW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

## Operation

```
DEST ←FPUControlWord;
```

## FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

## Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FSTENV/FNSTENV—Store x87 FPU Environment

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 9B D9 /6 | FSTENV *m14/28byte* | Valid | Valid | Store FPU environment to *m14byte* or *m28byte* after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions. |
| D9 /6 | FNSTENV* *m14/28byte* | Valid | Valid | Store FPU environment to *m14byte* or *m28byte* without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not. The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

The assembler issues two instructions for the FSTENV instruction (an FWAIT instruction followed by an FNSTENV instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTENV instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

## Operation

```
DEST[FPUControlWord] ←FPUControlWord;
DEST[FPUStatusWord] ←FPUStatusWord;
DEST[FPUTagWord] ←FPUTagWord;
DEST[FPUDataPointer] ←FPUDataPointer;
DEST[FPUInstructionPointer] ←FPUInstructionPointer;
DEST[FPULastInstructionOpcode] ←FPULastInstructionOpcode;
```

## FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FSTSW/FNSTSW—Store x87 FPU Status Word

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 9B DD /7 | FSTSW *m2byte* | Valid | Valid | Store FPU status word at *m2byte* after checking for pending unmasked floating-point exceptions. |
| 9B DF E0 | FSTSW AX | Valid | Valid | Store FPU status word in AX register after checking for pending unmasked floating-point exceptions. |
| DD /7 | FNSTSW* *m2byte* | Valid | Valid | Store FPU status word at *m2byte* without checking for pending unmasked floating-point exceptions. |
| DF E0 | FNSTSW* AX | Valid | Valid | Store FPU status word in AX register without checking for pending unmasked floating-point exceptions. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Stores the current value of the x87 FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See the section titled “Branching and Conditional Moves on FPU Condition Codes” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.) This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

The assembler issues two instructions for the FSTSW instruction (an FWAIT instruction followed by an FNSTSW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTSW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTSW instruction cannot be interrupted in this way on a Pentium 4, Intel Xeon, or P6 family processor.

## Operation

```
DEST ←FPUStatusWord;
```

## FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FSUB/FSUBP/FISUB—Subtract

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D8 /4 | FSUB *m32fp* | Valid | Valid | Subtract *m32fp* from ST(0) and store result in ST(0). |
| DC /4 | FSUB *m64fp* | Valid | Valid | Subtract *m64fp* from ST(0) and store result in ST(0). |
| D8 E0+i | FSUB ST(0), ST(i) | Valid | Valid | Subtract ST(i) from ST(0) and store result in ST(0). |
| DC E8+i | FSUB ST(i), ST(0) | Valid | Valid | Subtract ST(0) from ST(i) and store result in ST(i). |
| DE E8+i | FSUBP ST(i), ST(0) | Valid | Valid | Subtract ST(0) from ST(i), store result in ST(i), and pop register stack. |
| DE E9 | FSUBP | Valid | Valid | Subtract ST(0) from ST(1), store result in ST(1), and pop register stack. |
| DA /4 | FISUB *m32int* | Valid | Valid | Subtract *m32int* from ST(0) and store result in ST(0). |
| DE /4 | FISUB *m16int* | Valid | Valid | Subtract *m16int* from ST(0) and store result in ST(0). |

## Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a floating-point or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

Table 3-43 shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST –SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward −∞ mode, in which case the result is −0. This instruction also guarantees that +0 –(-0) = +0, and that -0 –(+0) = -0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

**Table 3-43. FSUB/FSUBP/FISUB Results**

| | | SRC | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | −∞ | −F or −I | −0 | +0 | +F or +I | +∞ | NaN |
| DEST | −∞ | * | −∞ | −∞ | −∞ | −∞ | −∞ | NaN |
| | −F | +∞ | ±F or ±0 | DEST | DEST | −F | −∞ | NaN |
| | −0 | +∞ | −SRC | ±0 | −0 | −SRC | −∞ | NaN |
| | +0 | +∞ | −SRC | +0 | ±0 | −SRC | −∞ | NaN |
| | +F | +∞ | +F | DEST | DEST | ±F or ±0 | −∞ | NaN |
| | +∞ | +∞ | +∞ | +∞ | +∞ | +∞ | * | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF Instruction = FISUB
    THEN
        DEST ←DEST –ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST ←DEST –SRC;
FI;
```

```
IF Instruction = FSUBP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Operand is an SNaN value or unsupported format. |
| | Operands are infinities of like sign. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #O | Result is too large for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FSUBR/FSUBRP/FISUBR—Reverse Subtract

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D8 /5 | FSUBR *m32fp* | Valid | Valid | Subtract ST(0) from *m32fp* and store result in ST(0). |
| DC /5 | FSUBR *m64fp* | Valid | Valid | Subtract ST(0) from *m64fp* and store result in ST(0). |
| D8 E8+i | FSUBR ST(0), ST(i) | Valid | Valid | Subtract ST(0) from ST(i) and store result in ST(0). |
| DC E0+i | FSUBR ST(i), ST(0) | Valid | Valid | Subtract ST(i) from ST(0) and store result in ST(i). |
| DE E0+i | FSUBRP ST(i), ST(0) | Valid | Valid | Subtract ST(i) from ST(0), store result in ST(i), and pop register stack. |
| DE E1 | FSUBRP | Valid | Valid | Subtract ST(1) from ST(0), store result in ST(1), and pop register stack. |
| DA /5 | FISUBR *m32int* | Valid | Valid | Subtract ST(0) from *m32int* and store result in ST(0). |
| DE /5 | FISUBR *m16int* | Valid | Valid | Subtract ST(0) from *m16int* and store result in ST(0). |

## Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a floating-point or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in

the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value (SRC – DEST = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward −∞ mode, in which case the result is −0. This instruction also guarantees that +0 – (−0) = +0, and that −0 – (+0) = −0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

**Table 3-44. FSUBR/FSUBRP/FISUBR Results**

| | | SRC | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | −∞ | −F or −I | −0 | +0 | +F or +I | +∞ | NaN |
| DEST | −∞ | * | +∞ | +∞ | +∞ | +∞ | +∞ | NaN |
| | −F | −∞ | ±F or ±0 | −DEST | −DEST | −F | +∞ | NaN |
| | −0 | −∞ | SRC | ±0 | +0 | SRC | +∞ | NaN |
| | +0 | −∞ | SRC | −0 | ±0 | SRC | +∞ | NaN |
| | +F | −∞ | −F | −DEST | −DEST | ±F or ±0 | +∞ | NaN |
| | +∞ | −∞ | −∞ | −∞ | −∞ | −∞ | * | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF Instruction = FISUBR
    THEN
        DEST ← ConvertToDoubleExtendedPrecisionFP(SRC) – DEST;
    ELSE (* Source operand is floating-point value *)
        DEST ← SRC – DEST; FI;
```

```
IF Instruction = FSUBRP
    THEN
        PopRegisterStack; FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Operand is an SNaN value or unsupported format. |
| | Operands are infinities of like sign. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #O | Result is too large for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# FTST—TEST

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 E4 | FTST | Valid | Valid | Compare ST(0) with 0.0. |

## Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

**Table 3-45. FTST Results**

| Condition | C3 | C2 | C0 |
|---|---|---|---|
| ST(0) > 0.0 | 0 | 0 | 0 |
| ST(0) < 0.0 | 0 | 0 | 1 |
| ST(0) = 0.0 | 1 | 0 | 0 |
| Unordered | 1 | 1 | 1 |

This instruction performs an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—ExamineModR/M" in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to "unordered" and the invalid operation exception is generated.

The sign of zero is ignored, so that (– 0.0 ←→ 0.0).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
CASE (relation of operands) OF
    Not comparable:  C3, C2, C0 ← 111;
    ST(0) > 0.0:     C3, C2, C0 ← 000;
    ST(0) < 0.0:     C3, C2, C0 ← 001;
    ST(0) = 0.0:     C3, C2, C0 ← 100;
ESAC;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; otherwise, set to 0. |
| C0, C2, C3 | See Table 3-45. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |

| | |
|---|---|
| #IA | The source operand is a NaN value or is in an unsupported format. |
| #D | The source operand is a denormal value. |

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| DD E0+i | FUCOM ST(i) | Valid | Valid | Compare ST(0) with ST(i). |
| DD E1 | FUCOM | Valid | Valid | Compare ST(0) with ST(1). |
| DD E8+i | FUCOMP ST(i) | Valid | Valid | Compare ST(0) with ST(i) and pop register stack. |
| DD E9 | FUCOMP | Valid | Valid | Compare ST(0) with ST(1) and pop register stack. |
| DA E9 | FUCOMPP | Valid | Valid | Compare ST(0) with ST(1) and pop register stack twice. |

## Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that –0.0 is equal to +0.0.

**Table 3-46. FUCOM/FUCOMP/FUCOMPP Results**

| Comparison Results* | C3 | C2 | C0 |
|---|---|---|---|
| ST0 > ST(i) | 0 | 0 | 0 |
| ST0 < ST(i) | 0 | 0 | 1 |
| ST0 = ST(i) | 1 | 0 | 0 |
| Unordered | 1 | 1 | 1 |

**NOTES:**

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see "FXAM—ExamineModR/M" in this chapter). The FUCOM/FUCOMP/FUCOMPP instructions perform the same operations as the FCOM/FCOMP/FCOMPP instructions. The only difference is that the FUCOM/FUCOMP/FUCOMPP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM/FCOMP/FCOMPP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM/FCOMP/FCOMPP instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
CASE (relation of operands) OF
    ST > SRC:       C3, C2, C0 ←000;
    ST < SRC:       C3, C2, C0 ←001;
    ST = SRC:       C3, C2, C0 ←100;
ESAC;

IF ST(0) or SRC = QNaN, but not SNaN or unsupported format
    THEN
        C3, C2, C0 ←111;
    ELSE (* ST(0) or SRC is SNaN or unsupported format *)
        #IA;
        IF FPUControlWord.IM = 1
            THEN
                C3, C2, C0 ←111;
        FI;
FI;

IF Instruction = FUCOMP
    THEN
        PopRegisterStack;
FI;

IF Instruction = FUCOMPP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| C0, C2, C3 | See Table 3-46. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |

| | |
|---|---|
| #IA | One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception. |
| #D | One or both operands are denormal values. |

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FXAM—ExamineModR/M

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 E5 | FXAM | Valid | Valid | Classify value or number in ST(0). |

## Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

**Table 3-47. FXAM Results**

| Class | C3 | C2 | C0 |
|---|---|---|---|
| Unsupported | 0 | 0 | 0 |
| NaN | 0 | 0 | 1 |
| Normal finite number | 0 | 1 | 0 |
| Infinity | 0 | 1 | 1 |
| Zero | 1 | 0 | 0 |
| Empty | 1 | 0 | 1 |
| Denormal number | 1 | 1 | 0 |

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
C1 ←sign bit of ST; (* 0 for positive, 1 for negative *)

CASE (class of value or number in ST(0)) OF
    Unsupported:C3, C2, C0 ←000;
    NaN:        C3, C2, C0 ←001;
    Normal:     C3, C2, C0 ←010;
    Infinity:   C3, C2, C0 ←011;
    Zero:       C3, C2, C0 ←100;
    Empty:      C3, C2, C0 ←101;
    Denormal:   C3, C2, C0 ←110;
ESAC;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Sign of value in ST(0). |
| C0, C2, C3 | See Table 3-47. |

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FXCH—Exchange Register Contents

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 C8+i | FXCH ST(i) | Valid | Valid | Exchange the contents of ST(0) and ST(i). |
| D9 C9 | FXCH | Valid | Valid | Exchange the contents of ST(0) and ST(1). |

## Description

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF (Number-of-operands) is 1
    THEN
        temp ←ST(0);
        ST(0) ←SRC;
        SRC ←temp;
    ELSE
        temp ←ST(0);
        ST(0) ←ST(1);
        ST(1) ←temp;
FI;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; otherwise, set to 1. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FXRSTOR—Restore x87 FPU, MMX , XMM, and MXCSR State

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F AE /1 | FXRSTOR *m512byte* | Valid | Valid | Restore the x87 FPU, MMX, XMM, and MXCSR register state from *m512byte*. |

## Description

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layout of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode with promoted operandsize, and the third format is for 64-bit mode with default operand size. Table 3-48 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-51 shows the layout of the 64-bit mode stat information when REX.W is set. Table 3-52 shows the layout of the 64-bit mode stat information when REX.W is clear.

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Table 3-48, Table 3-51, or Table 3-52. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

If the OSFXSR bit in control register CR4 is not set, the FXRSTOR instruction may not restore the states of the XMM and MXCSR registers. This behavior is implementation dependent.

If the MXCSR state contains an unmasked exception with a corresponding status flag also set, loading the register with the FXRSTOR instruction will not result in a SIMD floating-point error condition being generated. Only the next occurrence of this unmasked exception will result in the exception being generated.

Bits 16 through 32 of the MXCSR register are defined as reserved and should be set to 0. Attempting to write a 1 in any of these bits from the saved state image will result in a general protection exception (#GP) being generated.

## Operation

(x87 FPU, MMX, XMM7-XMM0, MXCSR) ←Load(SRC);

## x87 FPU and SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See alignment check exception [#AC] below.) |
| | For an attempt to set reserved bits in MXCSR. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CPUID.01H:EDX.FXSR[bit 24] = 0. |
| | If instruction is preceded by a LOCK prefix. |
| #AC | If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments). |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| | For an attempt to set reserved bits in MXCSR. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC | For unaligned memory reference. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | For an attempt to set reserved bits in MXCSR. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CPUID.01H:EDX.FXSR[bit 24] = 0. |
| | If instruction is preceded by a LOCK prefix. |
| #AC | If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments). |

# FXSAVE—Save x87 FPU, MMX Technology, SSE, and SSE2 State

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F AE /0 | FXSAVE *m512byte* | Valid | Valid | Save the x87 FPU, MMX, XMM, and MXCSR register state to *m512byte*. |

## Description

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512 byte region depends on whether the processor is operating in non-64-bit operating modes or 64-bit sub-mode of IA-32e mode. The operation of FXSAVE in non-64-bit modes are described first.

## Non-64-Bit Mode Operation

Table 3-48 shows the layout of the state information in memory when the processor is operating in legacy modes.

**Table 3-48. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region**

| 15 14 | 13 12 | 11 10 | 9 8 | 7 6 | 5 | 4 | 3 2 | 1 0 | |
|---|---|---|---|---|---|---|---|---|---|
| Rsrvd | CS | FPU IP | | FOP | | FTW | FSW | FCW | 0 |
| MXCSR_MASK | | MXCSR | | Rsrvd | DS | | FPU DP | | 16 |
| Reserved | | ST0/MM0 | | | | | | | 32 |
| Reserved | | ST1/MM1 | | | | | | | 48 |
| Reserved | | ST2/MM2 | | | | | | | 64 |
| Reserved | | ST3/MM3 | | | | | | | 80 |
| Reserved | | ST4/MM4 | | | | | | | 96 |
| Reserved | | ST5/MM5 | | | | | | | 112 |
| Reserved | | ST6/MM6 | | | | | | | 128 |
| Reserved | | ST7/MM7 | | | | | | | 144 |
| XMM0 | | | | | | | | | 160 |
| XMM1 | | | | | | | | | 176 |
| XMM2 | | | | | | | | | 192 |
| XMM3 | | | | | | | | | 208 |
| XMM4 | | | | | | | | | 224 |

**Table 3-48. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region (Contd.)**

| 15 14 | 13 12 | 11 10 | 9 8 | 7 6 | 5 4 | 3 2 | 1 0 | |
|---|---|---|---|---|---|---|---|---|
| XMM5 | | | | | | | | 240 |
| XMM6 | | | | | | | | 256 |
| XMM7 | | | | | | | | 272 |
| Reserved | | | | | | | | 288 |
| Reserved | | | | | | | | 304 |
| Reserved | | | | | | | | 320 |
| Reserved | | | | | | | | 336 |
| Reserved | | | | | | | | 352 |
| Reserved | | | | | | | | 368 |
| Reserved | | | | | | | | 384 |
| Reserved | | | | | | | | 400 |
| Reserved | | | | | | | | 416 |
| Reserved | | | | | | | | 432 |
| Reserved | | | | | | | | 448 |
| Reserved | | | | | | | | 464 |
| Reserved | | | | | | | | 480 |
| Reserved | | | | | | | | 496 |

The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-48 are defined in Table 3-49.

### Table 3-49. Field Definitions

| Field | Definition |
|---|---|
| FCW | x87 FPU Control Word (16 bits). See Figure 8-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU control word. |
| FSW | x87 FPU Status Word (16 bits). See Figure 8-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU status word. |
| FTW | x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs. See Figure 8-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU tag word. |
| FOP | x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU opcode field. |
| FPU IP | x87 FPU Instruction Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed:<br>32-bit mode — 32-bit IP offset.<br>16-bit mode — low 16 bits are IP offset; high 16 bits are reserved.<br>See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the x87 FPU instruction pointer. |
| CS | x87 FPU Instruction Pointer Selector (16 bits). |
| FPU DP | x87 FPU Instruction Operand (Data) Pointer Offset (32 bits). The contents of this field differ depending on the current addressing mode (32-bit or 16-bit) of the processor when the FXSAVE instruction was executed:<br>32-bit mode — 32-bit IP offset.<br>16-bit mode — low 16 bits are IP offset; high 16 bits are reserved. |
| | See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the x87 FPU operand pointer. |
| DS | x87 FPU Instruction Operand (Data) Pointer Selector (16 bits). |

**Table 3-49. Field Definitions (Contd.)**

| Field | Definition |
|---|---|
| MXCSR | MXCSR Register State (32 bits). See Figure 10-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the MXCSR register. If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save this register. This behavior is implementation dependent. |
| MXCSR_ MASK | MXCSR_MASK (32 bits). This mask can be used to adjust values written to the MXCSR register, ensuring that reserved bits are set to 0. Set the mask bits and flags in MXCSR to the mode of operation desired for SSE and SSE2 SIMD floating-point instructions. See "Guidelines for Writing to the MXCSR Register" in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for instructions for how to determine and use the MXCSR_MASK value. |
| ST0/MM0 through ST7/MM7 | x87 FPU or MMX technology registers. These 80-bit fields contain the x87 FPU data registers or the MMX technology registers, depending on the state of the processor prior to the execution of the FXSAVE instruction. If the processor had been executing x87 FPU instruction prior to the FXSAVE instruction, the x87 FPU data registers are saved; if it had been executing MMX instructions (or SSE or SSE2 instructions that operated on the MMX technology registers), the MMX technology registers are saved. When the MMX technology registers are saved, the high 16 bits of the field are reserved. |
| XMM0 through XMM7 | XMM registers (128 bits per field). If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save these registers. This behavior is implementation dependent. |

The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field (unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

| R7 | R6 | R5 | R4 | R3 | R2 | R1 | R0 |
|---|---|---|---|---|---|---|---|
| 11 | xx | xx | xx | 11 | 11 | 11 | 11 |

Here, 11B indicates empty stack elements and "xx" indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

| R7 | R6 | R5 | R4 | R3 | R2 | R1 | R0 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, the as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be used by an application program to pass a “clean” x87 FPU state to a procedure, since it retains the current state. To clean the x87 FPU state, an application must explicitly execute an FINIT instruction after an FXSAVE instruction to reinitialize the x87 FPU state.
- The format of the memory image saved with the FXSAVE instruction is the same regardless of the current addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management). This behavior differs from the FSAVE instructions, where the memory image format is different depending on the addressing mode and operating mode. Because of the different image formats, the memory image saved with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the stored data was not the contents of MMX technology registers) using Table 3-50.

### Table 3-50. Recreating FSAVE Format

| Exponent all 1’s | Exponent all 0’s | Fraction all 0’s | J and M bits | FTW valid bit | x87 FTW | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0x | 1 | Special | 10 |
| 0 | 0 | 0 | 1x | 1 | Valid | 00 |
| 0 | 0 | 1 | 00 | 1 | Special | 10 |
| 0 | 0 | 1 | 10 | 1 | Valid | 00 |
| 0 | 1 | 0 | 0x | 1 | Special | 10 |
| 0 | 1 | 0 | 1x | 1 | Special | 10 |
| 0 | 1 | 1 | 00 | 1 | Zero | 01 |
| 0 | 1 | 1 | 10 | 1 | Special | 10 |
| 1 | 0 | 0 | 1x | 1 | Special | 10 |
| 1 | 0 | 0 | 1x | 1 | Special | 10 |

### Table 3-50. Recreating FSAVE Format (Contd.)

| Exponent all 1's | Exponent all 0's | Fraction all 0's | J and M bits | FTW valid bit | x87 FTW |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 00 | 1 | Special 10 |
| 1 | 0 | 1 | 10 | 1 | Special 10 |
| For all legal combinations above. | | | | 0 | Empty 11 |

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0's.

## IA-32e Mode Operation

In compatibility sub-mode of IA-32e mode, legacy SSE registers, XMM0 through XMM7, are saved according to the legacy FXSAVE map. In 64-bit mode, all of the SSE registers, XMM0 through XMM15, are saved. But the layout of the 64-bit FXSAVE map has two flavors, depending on the value of the REX.W bit. The difference of these two flavors is in the FPU IP and FPU DP pointers. When REX.W = 0, the FPU IP is saved as CS with the 32 bit IP, and the FPU DP is saved as DS with the 32 bit DP. When REX.W = 1, the FPU IP and FPU DP are both 64 bit values without and segment selectors.

The IA-32e mode save formats are shown in Table 3-51 and Table 3-52 listed below.

### Table 3-51. Layout of the 64-bit-mode FXSAVE Map with Promoted OperandSize

| 15 14 | 13 12 | 11 10 | 9 8 | 7 6 | 5 | 4 | 3 2 | 1 0 | |
|---|---|---|---|---|---|---|---|---|---|
| FPU IP | | | | FOP | | FTW | FSW | FCW | **0** |
| MXCSR_MASK | | MXCSR | | FPU DP | | | | | **16** |
| Reserved | | ST0/MM0 | | | | | | | **32** |
| Reserved | | ST1/MM1 | | | | | | | **48** |
| Reserved | | ST2/MM2 | | | | | | | **64** |
| Reserved | | ST3/MM3 | | | | | | | **80** |
| Reserved | | ST4/MM4 | | | | | | | **96** |
| Reserved | | ST5/MM5 | | | | | | | **112** |
| Reserved | | ST6/MM6 | | | | | | | **128** |
| Reserved | | ST7/MM7 | | | | | | | **144** |
| XMM0 | | | | | | | | | **160** |
| XMM1 | | | | | | | | | **176** |

### Table 3-51. Layout of the 64-bit-mode FXSAVE Map with Promoted OperandSize (Contd.)

| 15 14 | 13 12 | 11 10 | 9 8 | 7 6 | 5 | 4 | 3 2 | 1 0 | |
|---|---|---|---|---|---|---|---|---|---|
| XMM2 | | | | | | | | | 192 |
| XMM3 | | | | | | | | | 208 |
| XMM4 | | | | | | | | | 224 |
| XMM5 | | | | | | | | | 240 |
| XMM6 | | | | | | | | | 256 |
| XMM7 | | | | | | | | | 272 |
| XMM8 | | | | | | | | | 288 |
| XMM9 | | | | | | | | | 304 |
| XMM10 | | | | | | | | | 320 |
| XMM11 | | | | | | | | | 336 |
| XMM12 | | | | | | | | | 352 |
| XMM13 | | | | | | | | | 368 |
| XMM14 | | | | | | | | | 384 |
| XMM15 | | | | | | | | | 400 |
| Reserved | | | | | | | | | 416 |
| Reserved | | | | | | | | | 432 |
| Reserved | | | | | | | | | 448 |
| Reserved | | | | | | | | | 464 |
| Reserved | | | | | | | | | 480 |
| Reserved | | | | | | | | | 496 |

### Table 3-52. Layout of the 64-bit-mode FXSAVE Map with Default OperandSize

| 15 14 | 13 12 | 11 10 | 9 8 | 7 6 | 5 | 4 | 3 2 | 1 0 | |
|---|---|---|---|---|---|---|---|---|---|
| Reserved | CS | FPU IP | | FOP | | FTW | FSW | FCW | 0 |
| MXCSR_MASK | | MXCSR | | Re-served | DS | | FPU DP | | 16 |
| Reserved | | ST0/MM0 | | | | | | | 32 |
| Reserved | | ST1/MM1 | | | | | | | 48 |
| Reserved | | ST2/MM2 | | | | | | | 64 |
| Reserved | | ST3/MM3 | | | | | | | 80 |

**Table 3-52. Layout of the 64-bit-mode FXSAVE Map with Default OperandSize (Contd.)**

| 15 14 | 13 12 | 11 10 | 9 8 | 7 6 | 5 4 | 3 2 | 1 0 | |
|---|---|---|---|---|---|---|---|---|
| Reserved | | | ST4/MM4 | | | | | 96 |
| Reserved | | | ST5/MM5 | | | | | 112 |
| Reserved | | | ST6/MM6 | | | | | 128 |
| Reserved | | | ST7/MM7 | | | | | 144 |
| XMM0 | | | | | | | | 160 |
| XMM1 | | | | | | | | 176 |
| XMM2 | | | | | | | | 192 |
| XMM3 | | | | | | | | 208 |
| XMM4 | | | | | | | | 224 |
| XMM5 | | | | | | | | 240 |
| XMM6 | | | | | | | | 256 |
| XMM7 | | | | | | | | 272 |
| XMM8 | | | | | | | | 288 |
| XMM9 | | | | | | | | 304 |
| XMM10 | | | | | | | | 320 |
| XMM11 | | | | | | | | 336 |
| XMM12 | | | | | | | | 352 |
| XMM13 | | | | | | | | 368 |
| XMM14 | | | | | | | | 384 |
| XMM15 | | | | | | | | 400 |
| Reserved | | | | | | | | 416 |
| Reserved | | | | | | | | 432 |
| Reserved | | | | | | | | 448 |
| Reserved | | | | | | | | 464 |
| Reserved | | | | | | | | 480 |
| Reserved | | | | | | | | 496 |

## Operation

```
IF 64-Bit Mode
    THEN
        IF REX.W = 1
```

```
            THEN
                DEST ←Save64BitPromotedFxsave(x87 FPU, MMX, XMM7-XMM0,
                MXCSR);
            ELSE
                DEST ←Save64BitDefaultFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);
        FI;
    ELSE
        DEST ←SaveLegacyFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);
FI;
```

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See the description of the alignment check exception [#AC] below.) |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1.<br>If CPUID.01H:EDX.FXSR[bit 24] = 0. |
| #UD | If the LOCK prefix is used. |
| #AC | If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments). |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment.<br>If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CPUID.01H:EDX.FXSR[bit 24] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC | For unaligned memory reference. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form.<br>If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #MF | If there is a pending x87 FPU exception. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1.<br>If CPUID.01H:EDX.FXSR[bit 24] = 0.<br>If the LOCK prefix is used. |
| #AC | If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments). |

## Implementation Note

The order in which the processor signals general-protection (#GP) and page-fault (#PF) exceptions when they both occur on an instruction boundary is given in Table 5-2 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This order vary for FXSAVE for different processor implementations.

# FXTRACT—Extract Exponent and Significand

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F4 | FXTRACT | Valid | Valid | Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack. |

## Description

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a floating-point value. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand's true (unbiased) exponent expressed as a floating-point value. (The operation performed by this instruction is a superset of the IEEE-recommended logb(*x*) function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in double extended-precision floating-point format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of $-\infty$ is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
TEMP ←Significand(ST(0));
ST(0) ←Exponent(ST(0));
TOP←TOP –1;
ST(0) ←TEMP;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow or overflow occurred. |

| | |
|---|---|
| #IA | Source operand is an SNaN value or unsupported format. |
| #Z | ST(0) operand is ±0. |
| #D | Source operand is a denormal value. |

### Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FYL2X—Compute y ∗ $\log_2$x

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F1 | FYL2X | Valid | Valid | Replace ST(1) with (ST(1) ∗ $\log_2$ST(0)) and pop the register stack. |

## Description

Computes (ST(1) ∗ $\log_2$ (ST(0))), stores the result in resister ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-53. FYL2X Results**

| | | ST(0) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | -∞ | -F | ±0 | -0 < -F < -1 | -1 | -F > -1 | +∞ | NaN |
| ST(1) | -∞ | * | * | +∞ | +∞ | * | -∞ | -∞ | NaN |
| | -F | * | * | ** | -F | -0 | -F | -∞ | NaN |
| | -0 | * | * | * | -0 | -0 | -0 | * | NaN |
| | +0 | * | * | * | -0 | +0 | -0 | * | NaN |
| | +F | * | * | ** | -F | -0 | -F | +∞ | NaN |
| | +∞ | * | * | -∞ | -∞ | * | +∞ | +∞ | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains ±0, the instruction returns ∞ with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base (b):

$$\log_b x \leftarrow (\log_2 b)^{-1} * \log_2 x$$

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

ST(1) ← ST(1) * $\log_2$ST(0);
PopRegisterStack;

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred.<br>Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Either operand is an SNaN or unsupported format.<br>Source operand in register ST(0) is a negative finite value (not -0). |
| #Z | Source operand in register ST(0) is ±0. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #O | Result is too large for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# FYL2XP1—Compute y * log2(x +1)

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| D9 F9 | FYL2XP1 | Valid | Valid | Replace ST(1) with ST(1) * $\log_2$(ST(0) + 1.0) and pop the register stack. |

## Description

Computes (ST(1) * $\log_2$(ST(0) +1.0)), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

$$-(1-\sqrt{2}/2))\text{to}(1-\sqrt{2}/2)$$

The source operand in ST(1) can range from −∞ to +∞. If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

**Table 3-54. FYL2XP1 Results**

| | | ST(0) | | | | |
|---|---|---|---|---|---|---|
| | | −(1 −(√2/ 2)) to −0 | −0 | +0 | +0 to +(1 −(√2/ 2)) | NaN |
| ST(1) | −∞ | +∞ | * | * | −∞ | NaN |
| | −F | +F | −0 | −0 | −F | NaN |
| | −0 | +0 | −0 | −0 | −0 | NaN |
| | +0 | −0 | −0 | +0 | +0 | NaN |
| | +F | −F | −0 | +0 | +F | NaN |
| | +∞ | −∞ | * | * | +∞ | NaN |
| | NaN | NaN | NaN | NaN | NaN | NaN |

**NOTES:**

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. For small epsilon (ε) values, more significant digits can be retained by using the FYL2XP1 instruction than by using (ε+1) as an argument to the FYL2X instruction. The (ε+1) expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following

equation is used to calculate the scale factor for a particular logarithm base, where n is the logarithm base desired for the result of the FYL2XP1 instruction:

$$\text{scale factor} \leftarrow \log_n 2$$

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
ST(1) ←ST(1) * log2(ST(0) +1.0);
PopRegisterStack;
```

## FPU Flags Affected

| | |
|---|---|
| C1 | Set to 0 if stack underflow occurred. |
| | Set if result was rounded up; cleared otherwise. |
| C0, C2, C3 | Undefined. |

## Floating-Point Exceptions

| | |
|---|---|
| #IS | Stack underflow occurred. |
| #IA | Either operand is an SNaN value or unsupported format. |
| #D | Source operand is a denormal value. |
| #U | Result is too small for destination format. |
| #O | Result is too large for destination format. |
| #P | Value cannot be represented exactly in destination format. |

## Protected Mode Exceptions

| | |
|---|---|
| #NM | CR0.EM[bit 2] or CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# HADDPD—Packed Double-FP Horizontal Add

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 7C /r | HADDPD *xmm1, xmm2/m128* | Valid | Valid | Horizontal add packed double-precision floating-point values from *xmm2/m128* to *xmm1*. |

## Description

Adds the double-precision floating-point values in the high and low quadwords of the destination operand and stores the result in the low quadword of the destination operand.

Adds the double-precision floating-point values in the high and low quadwords of the source operand and stores the result in the high quadword of the destination operand. See Figure 3-10.

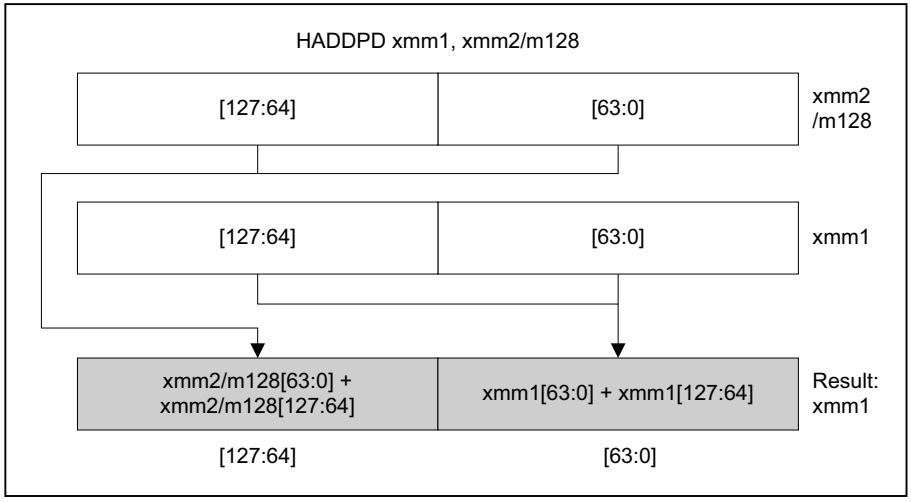



**Figure 3-10. HADDPD—Packed Double-FP Horizontal Add**

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
xmm1[63:0] = xmm1[63:0] + xmm1[127:64];
xmm1[127:64] = xmm2/m128[63:0] + xmm2/m128[127:64];
```

## Intel C/C++ Compiler Intrinsic Equivalent

HADDPD __m128d _mm_hadd_pd(__m128d a, __m128d b)

## Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

## Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

## Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID feature flag SSE3 is 0.<br>If the LOCK prefix is used. |

# HADDPS—Packed Single-FP Horizontal Add

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 7C /r | HADDPS *xmm1, xmm2/m128* | Valid | Valid | Horizontal add packed single-precision floating-point values from *xmm2/m128* to *xmm1*. |

## Description

Adds the single-precision floating-point values in the first and second dwords of the destination operand and stores the result in the first dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

Adds single-precision floating-point values in the first and second dword of the source operand and stores the result in the third dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the source operand and stores the result in the fourth dword of the destination operand. See Figure 3-11.

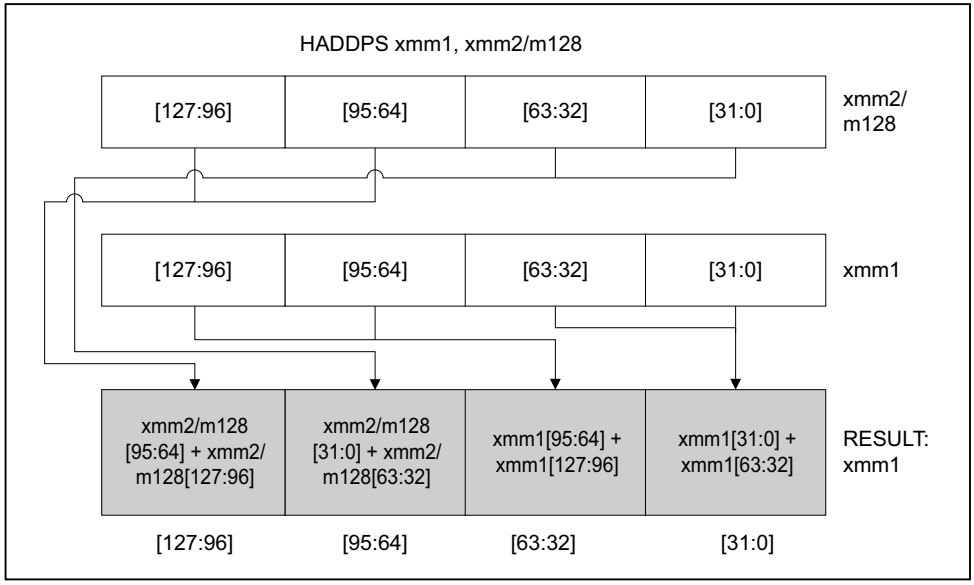



Figure 3-11. HADDPS—Packed Single-FP Horizontal Add

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
xmm1[31:0] = xmm1[31:0] + xmm1[63:32];
xmm1[63:32] = xmm1[95:64] + xmm1[127:96];
xmm1[95:64] = xmm2/m128[31:0] + xmm2/m128[63:32];
xmm1[127:96] = xmm2/m128[95:64] + xmm2/m128[127:96];
```

## Intel C/C++ Compiler Intrinsic Equivalent

HADDPS __m128 _mm_hadd_ps(__m128 a, __m128 b)

## Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

## Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID feature flag SSE3 is 0. |
| | If the LOCK prefix is used. |

# HLT—Halt

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F4 | HLT | Valid | Valid | Halt |

## Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
Enter Halt state;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

None.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# HSUBPD—Packed Double-FP Horizontal Subtract

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 7D /r | HSUBPD *xmm1, xmm2/m128* | Valid | Valid | Horizontal subtract packed double-precision floating-point values from *xmm2/m128* to *xmm1*. |

## Description

The HSUBPD instruction subtracts horizontally the packed DP FP numbers of both operands.

Subtracts the double-precision floating-point value in the high quadword of the destination operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand.

Subtracts the double-precision floating-point value in the high quadword of the source operand from the low quadword of the source operand and stores the result in the high quadword of the destination operand. See Figure 3-12.

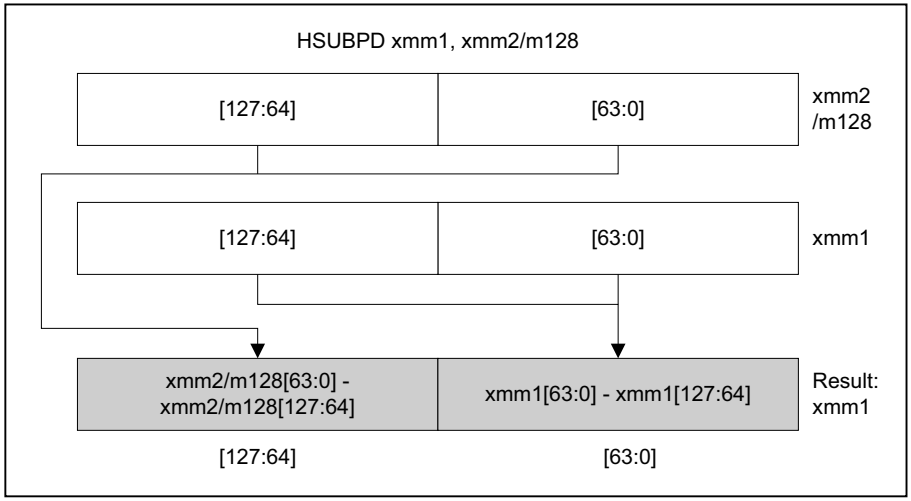



Figure 3-12. HSUBPD—Packed Double-FP Horizontal Subtract

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
xmm1[63:0] = xmm1[63:0] –xmm1[127:64];
xmm1[127:64] = xmm2/m128[63:0] –xmm2/m128[127:64];
```

## Intel C/C++ Compiler Intrinsic Equivalent

HSUBPD __m128d _mm_hsub_pd(__m128d a, __m128d b)

## Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

## Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

## Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |

If CPUID feature flag SSE3 is 0.
If the LOCK prefix is used.

# HSUBPS—Packed Single-FP Horizontal Subtract

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 7D /r | HSUBPS *xmm1, xmm2/m128* | Valid | Valid | Horizontal subtract packed single-precision floating-point values from *xmm2/m128* to *xmm1*. |

## Description

Subtracts the single-precision floating-point value in the second dword of the destination operand from the first dword of the destination operand and stores the result in the first dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

Subtracts the single-precision floating-point value in the second dword of the source operand from the first dword of the source operand and stores the result in the third dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

See Figure 3-13.

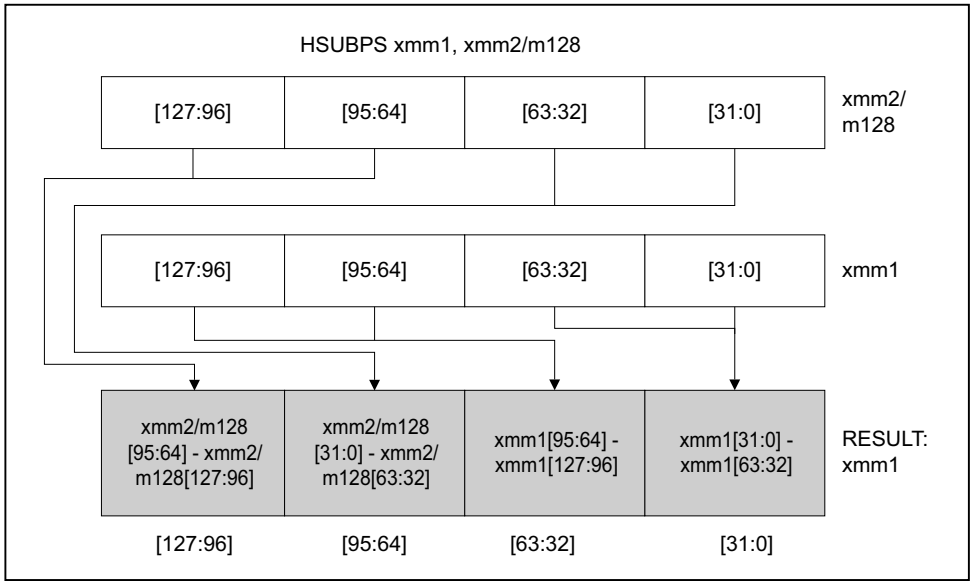



**Figure 3-13. HSUBPS—Packed Single-FP Horizontal Subtract**

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
xmm1[31:0] = xmm1[31:0] – xmm1[63:32];
xmm1[63:32] = xmm1[95:64] – xmm1[127:96];
xmm1[95:64] = xmm2/m128[31:0] – xmm2/m128[63:32];
xmm1[127:96] = xmm2/m128[95:64] – xmm2/m128[127:96];
```

## Intel C/C++ Compiler Intrinsic Equivalent

HSUBPS __m128 _mm_hsub_ps(__m128 a, __m128 b)

## Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

### Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 1). |
| #UD | If CR0.EM[bit 2] = 1. |
| | For an unmasked Streaming SIMD Extensions numeric exception (CR4.OSXMMEXCPT[bit 10] = 0). |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If the LOCK prefix is used. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

# IDIV—Signed Divide

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F6 /7 | IDIV *r/m8* | Valid | Valid | Signed divide AX by *r/m8*, with result stored in: AL ←Quotient, AH ← Remainder. |
| REX + F6 /7 | IDIV *r/m8** | Valid | N.E. | Signed divide AX by *r/m8*, with result stored in AL ←Quotient, AH ← Remainder. |
| F7 /7 | IDIV *r/m16* | Valid | Valid | Signed divide DX:AX by *r/m16*, with result stored in AX ←Quotient, DX ← Remainder. |
| F7 /7 | IDIV *r/m32* | Valid | Valid | Signed divide EDX:EAX by *r/m32*, with result stored in EAX ←Quotient, EDX ← Remainder. |
| REX.W + F7 /7 | IDIV *r/m64* | Valid | N.E. | Signed divide RDX:RAX by *r/m64*, with result stored in RAX ←Quotient, RDX ← Remainder. |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Divides the (signed) value in the AX, DX:AX, or EDX:EAX (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor).

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the signed value in RDX:RAX by the source operand. RAX contains a 64-bit quotient; RDX contains a 64-bit remainder.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-55.

**Table 3-55. IDIV Results**

| Operand Size | Dividend | Divisor | Quotient | Remainder | Quotient Range |
|---|---|---|---|---|---|
| Word/byte | AX | r/m8 | AL | AH | −128 to +127 |
| Doubleword/word | DX:AX | r/m16 | AX | DX | −32,768 to +32,767 |
| Quadword/doubleword | EDX:EAX | r/m32 | EAX | EDX | $-2^{31}$ to $2^{32}-1$ |
| Doublequadword/ quadword | RDX:RAX | r/m64 | RAX | RDX | $-2^{63}$ to $2^{64}-1$ |

## Operation

```
IF SRC = 0
    THEN #DE; (* Divide error *)
FI;

IF OperandSize = 8 (* Word/byte operation *)
    THEN
        temp ←AX / SRC; (* Signed division *)
        IF (temp > 7FH) or (temp < 80H)
        (* If a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* Divide error *)
            ELSE
                AL ←temp;
                AH ←AX SignedModulus SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp ←DX:AX / SRC; (* Signed division *)
            IF (temp > 7FFFH) or (temp < 8000H)
            (* If a positive result is greater than 7FFFH
            or a negative result is less than 8000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    AX ←temp;
                    DX ←DX:AX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
            temp ←EDX:EAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFH) or (temp < 80000000H)
            (* If a positive result is greater than 7FFFFFFFH
            or a negative result is less than 80000000H *)
```

```
                THEN
                    #DE; (* Divide error *)
                ELSE
                    EAX ← temp;
                    EDX ← EDXE:AX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
            temp ← RDX:RAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFFFH) or (temp < 8000000000000000H)
            (* If a positive result is greater than 7FFFFFFFFFFFH
            or a negative result is less than 8000000000000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    RAX ← temp;
                    RDX ← RDE:RAX SignedModulus SRC;
            FI;
        FI;
FI;
```

## Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #DE | If the source operand (divisor) is 0. |
| | The signed result (quotient) is too large for the destination. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #DE | If the source operand (divisor) is 0. |
| | The signed result (quotient) is too large for the destination. |

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #DE | If the source operand (divisor) is 0.<br><br>The signed result (quotient) is too large for the destination. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #DE | If the source operand (divisor) is 0<br><br>If the quotient is too large for the designated register. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## IMUL—Signed Multiply

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F6 /5 | IMUL *r/m8** | Valid | Valid | AX←AL ∗ *r/m* byte. |
| F7 /5 | IMUL *r/m16* | Valid | Valid | DX:AX ←AX ∗ *r/m* word. |
| F7 /5 | IMUL *r/m32* | Valid | Valid | EDX:EAX ←EAX ∗ *r/m*32. |
| REX.W + F7 /5 | IMUL *r/m64* | Valid | N.E. | RDX:RAX ←RAX ∗ *r/m*64. |
| 0F AF /*r* | IMUL *r16, r/m16* | Valid | Valid | word register ←word register ∗ *r/m*16. |
| 0F AF /*r* | IMUL *r32, r/m32* | Valid | Valid | doubleword register ← doubleword register ∗ *r/m32.* |
| REX.W + 0F AF /*r* | IMUL *r64, r/m64* | Valid | N.E. | Quadword register ←Quadword register ∗ *r/m64.* |
| 6B /*r ib* | IMUL *r16, r/m16, imm8* | Valid | Valid | word register ←*r/m16* ∗ sign-extended immediate byte. |
| 6B /*r ib* | IMUL *r32, r/m32, imm8* | Valid | Valid | doubleword register ←*r/m32* ∗ sign-extended immediate byte. |
| REX.W + 6B /*r ib* | IMUL *r64, r/m64, imm8* | Valid | N.E. | Quadword register ←*r/m64* ∗ sign-extended immediate byte. |
| 6B /*r ib* | IMUL *r16, imm8* | Valid | Valid | word register ←word register ∗ sign-extended immediate byte. |
| 6B /*r ib* | IMUL *r32, imm8* | Valid | Valid | doubleword register ← doubleword register ∗ sign-extended immediate byte. |
| REX.W + 6B /*r ib* | IMUL *r64, imm8* | Valid | N.E. | Quadword register ←Quadword register ∗ sign-extended immediate byte. |
| 69 /*r iw* | IMUL *r16, r/m16, imm16* | Valid | Valid | word register ←*r/m16* ∗ immediate word. |
| 69 /*r id* | IMUL *r32, r/m32, imm32* | Valid | Valid | doubleword register ←*r/m32* ∗ immediate doubleword. |
| REX.W + 69 /*r id* | IMUL *r64, r/m64, imm32* | Valid | N.E. | Quadword register ←*r/m64* ∗ immediate doubleword. |
| 69 /*r iw* | IMUL *r16, imm16* | Valid | Valid | word register ←*r/m16* ∗ immediate word. |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 69 /r id | IMUL r32, imm32 | Valid | Valid | doubleword register ←r/m32 * immediate doubleword. |
| REX.W + 69 /r id | IMUL r64, imm32 | Valid | N.E. | Quadword register ←r/m64 * immediate doubleword. |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.
- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The product is then stored in the destination operand location.
- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The product is then stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when significant bit (including the sign bit) are carried into the upper half of the result. The CF and OF flags are cleared when the result (including the sign bit) fits exactly in the lower half of the result.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are

signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- One-operand form —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. If the source operand is an immediate, it is sign extended to 64 bits. The destination operand is promoted to 64 bits.
- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits.

## Operation

```
IF (NumberOfOperands = 1)
    THEN IF (OperandSize = 8)
        THEN
            AX ←AL * SRC  (* Signed multiplication *)
            IF AL = AX
                THEN CF ←0; OF ←0;
                ELSE CF ←1; OF ←1; FI;
        ELSE IF OperandSize = 16
            THEN
                DX:AX ←AX * SRC  (* Signed multiplication *)
                IF sign_extend_to_32 (AX) = DX:AX
                    THEN CF ←0; OF ←0;
                    ELSE CF ←1; OF ←1; FI;
            ELSE IF OperandSize = 32
                THEN
                    EDX:EAX ←EAX * SRC  (* Signed multiplication *)
                    IF EAX = EDX:EAX
                        THEN CF ←0; OF ←0;
                        ELSE CF ←1; OF ←1; FI;
                ELSE (* OperandSize = 64 *)
                    RDX:RAX ←RAX * SRC  (* Signed multiplication *)
                    IF RAX = RDX:RAX
                        THEN CF ←0; OF ←0;
                        ELSE CF ←1; OF ←1; FI;
                FI;
        FI;
```

```
    ELSE IF (NumberOfOperands = 2)
        THEN
            temp ←DEST * SRC (* Signed multiplication; temp is double DEST size *)
            DEST ←DEST * SRC (* Signed multiplication *)
            IF temp ≠ DEST
                THEN CF ←1; OF ←1;
                ELSE CF ←0; OF ←0; FI;
        ELSE (* NumberOfOperands = 3 *)
            DEST ←SRC1 * SRC2 (* Signed multiplication *)
            temp ←SRC1 * SRC2 (* Signed multiplication; temp is double SRC1 size *)
            IF temp ≠ DEST
                THEN CF ←1; OF ←1;
                ELSE CF ←0; OF ←0; FI;
    FI;
FI;
```

## Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# IN—Input from Port

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| E4 *ib* | IN AL, i*mm8* | Valid | Valid | Input byte from *imm8* I/O port address into AL. |
| E5 *ib* | IN AX, i*mm8* | Valid | Valid | Input word from *imm8* I/O port address into AX. |
| E5 *ib* | IN EAX, i*mm8* | Valid | Valid | Input dword from *imm8* I/O port address into EAX. |
| EC | IN AL,DX | Valid | Valid | Input byte from I/O port in DX into AL. |
| ED | IN AX,DX | Valid | Valid | Input word from I/O port in DX into AX. |
| ED | IN EAX,DX | Valid | Valid | Input doubleword from I/O port in DX into EAX. |

## Description

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size. At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 13, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
```

```
                DEST ←SRC; (* Read from selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤IOPL *)
        DEST ←SRC; (* Read from selected I/O port *)
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If any of the I/O permission bits in the TSS for the I/O port being accessed is 1. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. |
| #UD | If the LOCK prefix is used. |

# INC—Increment by 1

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| FE /0 | INC *r/m8* | Valid | Valid | Increment *r/m* byte by 1. |
| REX + FE /0 | INC *r/m8** | Valid | N.E. | Increment *r/m* byte by 1. |
| FF /0 | INC *r/m16* | Valid | Valid | Increment *r/m* word by 1. |
| FF /0 | INC *r/m32* | Valid | Valid | Increment *r/m* doubleword by 1. |
| REX.W + FF /0 | INC *r/m64* | Valid | N.E. | Increment *r/m* quadword by 1. |
| 40+ *rw*** | INC *r16* | N.E. | Valid | Increment word register by 1. |
| 40+ *rd* | INC *r32* | N.E. | Valid | Increment doubleword register by 1. |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** 40H through 47H are REX prefixes in 64-bit mode.

## Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, INC r16 and INC r32 are not encodable (because opcodes 40H through 47H are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

## Operation

DEST ←DEST +1;

## AFlags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination operand is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULLsegment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used but the destination is not a memory operand. |

# INS/INSB/INSW/INSD—Input from Port to String

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 6C | INS *m8*, DX | Valid | Valid | Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.* |
| 6D | INS *m16*, DX | Valid | Valid | Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1] |
| 6D | INS *m32*, DX | Valid | Valid | Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1] |
| 6C | INSB | Valid | Valid | Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI.[1] |
| 6D | INSW | Valid | Valid | Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1] |
| 6D | INSD | Valid | Valid | Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1] |

**NOTES:**

* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

## Description

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:DI, ES:EDI or the RDI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The ES segment cannot be overridden with a segment override prefix.) The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the INS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand must be “DX,” and the destination operand should be a symbol that indicates the size of the I/O port and the destination address. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the

destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the INS instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the INS instructions. Here also DX is assumed by the processor to be the source operand and ES:(E)DI is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: INSB (byte), INSW (word), or INSD (doubleword).

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the DI/EDI/RDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 13, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, default address size is 64 bits, 32 bit address size is supported using the prefix 67H. The address of the memory destination is specified by RDI or EDI. 16-bit address size is not supported in 64-bit mode. The operand size is not promoted.

## Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST ←SRC; (* Read from I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL IOPL *)
        DEST ←SRC; (* Read from I/O port *)
FI;
```

Non-64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E)DI ←(E)DI +1;
        ELSE (E)DI ←(E)DI – 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI ←(E)DI +2;
            ELSE (E)DI ←(E)DI – 2; FI;
        ELSE (* Doubleword transfer *)
            THEN IF DF = 0
                THEN (E)DI ←(E)DI +4;
                ELSE (E)DI ←(E)DI – 4; FI;
        FI;
FI;
```

FI64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E|R)DI ←(E|R)DI +1;
        ELSE (E|R)DI ←(E|R)DI – 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI ←(E)DI +2;
            ELSE (E)DI ←(E)DI – 2; FI;
        ELSE (* Doubleword transfer *)
            THEN IF DF = 0
                THEN (E|R)DI ←(E|R)DI +4;
                ELSE (E|R)DI ←(E|R)DI – 4; FI;
        FI;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.<br>If the destination is located in a non-writable segment.<br>If an illegal memory operand effective address in the ES segments is given. |
| #PF(fault-code) | If a page fault occurs. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If any of the I/O permission bits in the TSS for the I/O port being accessed is 1. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.<br>If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# INT *n*/INTO/INT 3—Call to Interrupt Procedure

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| CC | INT 3 | Valid | Valid | Interrupt 3—trap to debugger. |
| CD *ib* | INT *imm8* | Valid | Valid | Interrupt vector number specified by immediate byte. |
| CE | INTO | Invalid | Valid | Interrupt 4—if overflow flag is 1. |

## Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled “Interrupts and Exceptions” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). The destination operand specifies an interrupt vector number from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each interrupt vector number provides an index to a gate descriptor in the IDT. The first 32 interrupt vector numbers are reserved by Intel for system use. Some of these interrupts are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), interrupt vector number 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1.

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code). To further support its function as a debug breakpoint, the interrupt generated with the CC opcode also differs from the regular software interrupts as follows:

- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level.

Note that the “normal” 2-byte opcode for INT 3 (CD03) does not have these special features. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.

The action of the INT *n* instruction (including the INTO and INT 3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT *n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled

with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The interrupt vector number specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the “Operation” section for this instruction (except #GP).

**Table 3-56. Decision Table**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **PE** | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **VM** | – | – | – | – | – | 0 | 1 | 1 |
| **IOPL** | – | – | – | – | – | – | <3 | =3 |
| **DPL/CPL RELATIONSHIP** | – | DPL< CPL | – | DPL> CPL | DPL= CPL or C | DPL< CPL & NC | – | – |
| **INTERRUPT TYPE** | – | S/W | – | – | – | – | – | – |
| **GATE TYPE** | – | – | Task | Trap or Interrupt | Trap or Interrupt | Trap or Interrupt | Trap or Interrupt | Trap or Interrupt |
| **REAL-ADDRESS-MODE** | Y | | | | | | | |
| **PROTECTED-MODE** | | Y | Y | Y | Y | Y | Y | Y |
| **TRAP-OR-INTERRUPT-GATE** | | | | Y | Y | Y | Y | Y |
| **INTER-PRIVILEGE-LEVEL-INTERRUPT** | | | | | | Y | | |
| **INTRA-PRIVILEGE-LEVEL-INTERRUPT** | | | | | Y | | | |
| **INTERRUPT-FROM-VIRTUAL-8086-MODE** | | | | | | | | Y |
| **TASK-GATE** | | | Y | | | | | |
| **#GP** | | Y | | Y | | | Y | |

**NOTES:**

– Don’t Care.

Y Yes, action taken.

Blank Action not taken.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

## Operation

The following operational description applies not only to the INT *n* and INTO instructions, but also to external interrupts and exceptions.

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE (* PE = 1 *)
        IF (VM = 1 and IOPL < 3 AND INT n)
            THEN
                #GP(0);
            ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                IF (IA32_EFER.LMA = 0)
                    THEN (* Protected mode, or virtual-8086 mode interrupt *)
                        GOTO PROTECTED-MODE;
                    ELSE (* IA-32e mode interrupt *)
                        GOTO IA-32e-MODE;
                FI;
        FI;
FI;
REAL-ADDRESS-MODE:
    IF ((vector_number * 4) +3) is not within IDT limit
        THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
        THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ←0; (* Clear interrupt flag *)
    TF ←0; (* Clear trap flag *)
    AC ←0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed *)
```

```
    CS ←IDT(Descriptor (vector_number * 4), selector));
    EIP ←IDT(Descriptor (vector_number * 4), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
    IF ((vector_number * 8) +7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP((vector_number * 8) +2 +EXT); FI;
        (* EXT is bit 0 in error code *)
    IF software interrupt (* Generated by INT n, INT 3, or INTO *)
        THEN
            IF gate descriptor DPL < CPL
                THEN #GP((vector_number * 8) +2 ); FI;
                (* PE = 1, DPL<CPL, software interrupt *)
    FI;
    IF gate not present
        THEN #NP((vector_number * 8) +2 +EXT); FI;
    IF task gate (* Specified in the selected interrupt table descriptor *)
        THEN GOTO TASK-GATE;
        ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
    FI;
END;
IA-32e-MODE:
    IF ((vector_number * 16) +15) is not in IDT limits
    or selected IDT descriptor is not an interrupt-, or trap-gate type
        THEN #GP((vector_number * 16) +2 +EXT); FI;
        (* EXT is bit 0 in error code *)
    IF software interrupt (* Generated by INT n, INT 3, but not INTO *)
        THEN
            IF gate descriptor DPL < CPL
                THEN #GP((vector_number * 16) +2 ); FI;
                (* PE = 1, DPL < CPL, software interrupt *)
        ELSE (* Generated by INTO *)
            THEN #UD;
    FI;
    IF gate not present
        THEN #NP((vector_number * 16) +2 +EXT); FI;
    IF ((vector_number * 16)[IST] ≠ 0)
        NewRSP ←TSS[ISTx]; FI;
    GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read segment selector in task gate (IDT descriptor);
        IF local/global bit is set to local
```

```
        or index not within GDT limits
            THEN #GP(TSS selector); FI;
        Access TSS descriptor in GDT;
        IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
            THEN #GP(TSS selector); FI;
        IF TSS not present
            THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF interrupt caused by fault with error code
        THEN
            IF stack limit does not allow push of error code
                THEN #SS(0); FI;
            Push(error code);
    FI;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
TRAP-OR-INTERRUPT-GATE:
    Read segment selector for trap or interrupt gate (IDT descriptor);
    IF segment selector for code segment is NULL
        THEN #GP(0H + EXT); FI; (* NULL selector with EXT flag set *)
    IF segment selector is not within its descriptor table limits
        THEN #GP(selector + EXT); FI;
    Read trap or interrupt handler descriptor;
    IF descriptor does not indicate a code segment
    or code segment descriptor DPL > CPL
        THEN #GP(selector + EXT); FI;
    IF trap or interrupt gate segment is not present,
        THEN #NP(selector + EXT); FI;
    IF code segment is non-conforming and DPL < CPL
        THEN
            IF VM = 0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE = 1, interrupt or trap gate, nonconforming
                    code segment, DPL < CPL, VM = 0 *)
                ELSE (* VM = 1 *)
                    IF code segment DPL ≠ 0
                        THEN #GP; (new code segment selector);
                    GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
                    (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
            FI;
        ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
```

```
            IF VM = 1
                THEN #GP(new code segment selector); FI;
            IF code segment is conforming or code segment DPL = CPL
                THEN
                    GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                ELSE
                    #GP(CodeSegmentSelector +EXT);
                    (* PE = 1, interrupt or trap gate, nonconforming
                    code segment, DPL > CPL *)
            FI;
    FI;
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
    (* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
    (* Check segment selector and descriptor for stack of new privilege level in current TSS *)
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ←(new code segment DPL * 8) +4;
            IF (TSSstackAddress +7) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ←TSSstackAddress +4;
            NewESP ←stack address;
        ELSE
            IF current TSS is 16-bit TSS
                THEN(* TSS is 16-bit *)
                    TSSstackAddress ←(new code segment DPL * 4) +2
                    IF (TSSstackAddress +4) > TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewESP ←TSSstackAddress;
                    NewSS ←TSSstackAddress +2;
                ELSE (* TSS is 64-bit *)
                    NewESP ←TSS[RSP FOR NEW TARGET DPL];
                    NewSS ←0;
            FI;
    FI;
    IF segment selector is NULL
        THEN #TS(EXT); FI;
    IF segment selector index is not within its descriptor table limits
    or segment selector's RPL ≠ DPL of code segment,
        THEN #TS(SS selector +EXT); FI;
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        Read segment descriptor for stack segment in GDT or LDT;
        IF stack segment DPL ≠ DPL of code segment,
```

```
        or stack segment does not indicate writable data segment
            THEN #TS(SS selector +EXT); FI;
        IF stack segment not present
            THEN #SS(SS selector +EXT); FI;
    FI
    IF 32-bit gate
            THEN
                IF new stack does not have room for 24 bytes (error code pushed)
                or 20 bytes (no error code pushed)
                    THEN #SS(segment selector +EXT); FI;
            FI
        ELSE
            IF 16-bit gate
                THEN
                    IF new stack does not have room for 12 bytes (error code pushed)
                    or 10 bytes (no error code pushed);
                    THEN #SS(segment selector +EXT); FI;
            ELSE (* 64-bit gate*)
                IF StackAddress is non-canonical
                    THEN #SS(0);FI;
        FI;
    FI;
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer is not within code segment limits
                THEN #GP(0); FI;
            SS:ESP ←TSS(NewSS:NewESP);
                (* Segment descriptor information also loaded *)
        ELSE
            IF instruction pointer points to non-canonical address
                THEN #GP(0); FI:
    FI;
    IF 32-bit gate
        THEN
            CS:EIP ←Gate(CS:EIP); (* Segment descriptor information also loaded *)
        ELSE
            IF 16-bit gate
                THEN
                    CS:IP ←Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                ELSE (* 64-bit gate *)
                    CS:RIP ←Gate(CS:RIP);
                    (* Segment descriptor information also loaded *)
            FI;
```

```
    FI;
    IF 32-bit gate
            THEN
                Push(far pointer to old stack);
                (* Old SS and ESP, 3 words padded to 4 *)
                Push(EFLAGS);
                Push(far pointer to return instruction);
                (* Old CS and EIP, 3 words padded to 4 *)
                Push(ErrorCode); (* If needed, 4 bytes *)
            ELSE
                IF 16-bit gate
                    THEN
                        Push(far pointer to old stack);
                        (* Old SS and SP, 2 words *)
                        Push(EFLAGS(15-0]);
                        Push(far pointer to return instruction);
                        (* Old CS and IP, 2 words *)
                        Push(ErrorCode); (* If needed, 2 bytes *)
                    ELSE (* 64-bit gate *)
                        Push(far pointer to old stack);
                        (* Old SS and SP, each an 8-byte push *)
                        Push(RFLAGS); (* 8-byte push *)
                        Push(far pointer to return instruction);
                        (* Old CS and RIP, each an 8-byte push *)
                        Push(ErrorCode); (* If needed, 8-bytes *)
            FI;
    FI;
    CPL ←CodeSegmentDescriptor(DPL);
    CS(RPL) ←CPL;
    IF interrupt gate
        THEN IF ←0 (* Interrupt flag set to 0: disabled *); FI;
    TF ←0;
    VM ←0;
    RF ←0;
    NT ←0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
    (* Check segment selector and descriptor for privilege level 0 stack in current TSS *)
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ←(new code segment DPL * 8) +4;
            IF (TSSstackAddress +7) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ←TSSstackAddress +4;
```

```
            NewESP ←stack address;
        ELSE (* TSS is 16-bit *)
            TSSstackAddress ←(new code segment DPL * 4) +2;
            IF (TSSstackAddress +4) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewESP ←TSSstackAddress;
            NewSS ←TSSstackAddress +2;
FI;
IF segment selector is NULL
    THEN #TS(EXT); FI;
IF segment selector index is not within its descriptor table limits
or segment selector's RPL ≠ DPL of code segment
    THEN #TS(SS selector +EXT); FI;
Access segment descriptor for stack segment in GDT or LDT;
IF stack segment DPL ≠ DPL of code segment,
or stack segment does not indicate writable data segment
    THEN #TS(SS selector +EXT); FI;
IF stack segment not present
    THEN #SS(SS selector +EXT); FI;
IF 32-bit gate
    THEN
            IF new stack does not have room for 40 bytes (error code pushed)
            or 36 bytes (no error code pushed)
                THEN #SS(segment selector +EXT); FI;
    ELSE IF 16-bit gate
        THEN
            IF new stack does not have room for 20 bytes (error code pushed)
            or 18 bytes (no error code pushed)
                THEN #SS(segment selector +EXT); FI;
        ELSE (* 64-bit gate*)
            IF StackAddress is non-canonical
                THEN #SS(0);
        FI;
FI;
IF instruction pointer is not within code segment limits
    THEN #GP(0); FI;
tempEFLAGS ←EFLAGS;
VM ←0;
TF ←0;
RF ←0;
NT ←0;
IF service through interrupt gate
    THEN IF = 0; FI;
```

```
    TempSS ←SS;
    TempESP ←ESP;
    SS:ESP ←TSS(SS0:ESP0); (* Change to level 0 stack segment *)
    (* Following pushes are 16 bits for 16-bit gate and 32 bits for 32-bit gates;
    Segment selector pushes in 32-bit mode are padded to two words *)
    Push(GS);
    Push(FS);
    Push(DS);
    Push(ES);
    Push(TempSS);
    Push(TempESP);
    Push(TempEFlags);
    Push(CS);
    Push(EIP);
    GS ←0; (* Segment registers NULLified, invalid in protected mode *)
    FS ←0;
    DS ←0;
    ES ←0;
    CS ←Gate(CS);
    IF OperandSize = 32
        THEN
            EIP ←Gate(instruction pointer);
        ELSE (* OperandSize is 16 *)
            EIP ←Gate(instruction pointer) AND 0000FFFFH;
    FI;
    (* Start execution of new routine in Protected Mode *)
END;
INTRA-PRIVILEGE-LEVEL-INTERRUPT:
    (* PE = 1, DPL = CPL or conforming segment *)
    IF 32-bit gate and IA32_EFER.LMA = 0
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
            or 12 bytes (no error code pushed)
                THEN #SS(0); FI;
        ELSE IF 16-bit gate
            IF current stack does not have room for 8 bytes (error code pushed)
            or 6 bytes (no error code pushed)
                THEN #SS(0); FI;
        ELSE (* 64-bit gate*)
                IF StackAddress is non-canonical
                    THEN #SS(0);
        FI;
    FI;
```

```
    IF instruction pointer not within code segment limit
        THEN #GP(0); FI;
    IF 32-bit gate
        THEN
            Push (EFLAGS);
            Push (far pointer to return instruction); (* 3 words padded to 4 *)
            CS:EIP ←Gate(CS:EIP); (* Segment descriptor information also loaded *)
            Push (ErrorCode); (* If any *)
        ELSE
            IF 16-bit gate
                THEN
                    Push (FLAGS);
                    Push (far pointer to return location); (* 2 words *)
                    CS:IP ←Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                    Push (ErrorCode); (* If any *)
                ELSE (* 64-bit gate*)
                    Push(far pointer to old stack);
                    (* Old SS and SP, each an 8-byte push *)
                    Push(RFLAGS); (* 8-byte push *)
                    Push(far pointer to return instruction);
                    (* Old CS and RIP, each an 8-byte push *)
                    Push(ErrorCode); (* If needed, 8 bytes *)
                    CS:RIP ←GATE(CS:RIP);
                    (* Segment descriptor information also loaded *)
            FI;
    FI;
    CS(RPL) ←CPL;
    IF interrupt gate
        THEN IF ←0; FI; (* Interrupt flag set to 0: disabled *)
    TF ←0;
    NT ←0;
    VM ←0;
    RF ←0;
END;
```

## Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the “Operation” section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task’s TSS.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits. |
| #GP(selector) | If the segment selector in the interrupt-, trap-, or task gate is NULL. |
| | If an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits. |
| | If the interrupt vector number is outside the IDT limits. |
| | If an IDT descriptor is not an interrupt-, trap-, or task-descriptor. |
| | If an interrupt is generated by the INT *n*, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL. |
| | If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment. |
| | If the segment selector for a TSS has its local/global bit set for local. |
| | If a TSS segment descriptor specifies that the TSS is busy or not available. |
| #SS(0) | If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs. |
| #SS(selector) | If the SS register is being loaded and the segment pointed to is marked not present. |
| | If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs. |
| #NP(selector) | If code segment, interrupt-, trap-, or task gate, or TSS is not present. |
| #TS(selector) | If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate. |
| | If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate. |
| | If the stack segment selector in the TSS is NULL. |
| | If the stack segment for the TSS is not a writable data segment. |
| | If segment-selector index for stack segment is outside descriptor table limits. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the interrupt vector number is outside the IDT limits. |
| #SS | If stack limit violation on push. |
| | If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | (For INT *n,* INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3. |
| | If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits. |
| #GP(selector) | If the segment selector in the interrupt-, trap-, or task gate is NULL. |
| | If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits. |
| | If the interrupt vector number is outside the IDT limits. |
| | If an IDT descriptor is not an interrupt-, trap-, or task-descriptor. |
| | If an interrupt is generated by the INT *n* instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL. |
| | If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment. |
| | If the segment selector for a TSS has its local/global bit set for local. |
| #SS(selector) | If the SS register is being loaded and the segment pointed to is marked not present. |
| | If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment. |
| #NP(selector) | If code segment, interrupt-, trap-, or task gate, or TSS is not present. |
| #TS(selector) | If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate. |
| | If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate. |
| | If the stack segment selector in the TSS is NULL. |

| | |
|---|---|
| | If the stack segment for the TSS is not a writable data segment. |
| | If segment-selector index for stack segment is outside descriptor table limits. |
| #PF(fault-code) | If a page fault occurs. |
| #BP | If the INT 3 instruction is executed. |
| #OF | If the INTO instruction is executed and the OF flag is set. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical. |
| #GP(selector) | If the segment selector in the 64-bit interrupt or trap gate is NULL. |
| | If the interrupt vector number is outside the IDT limits. |
| | If the interrupt vector number points to a gate which is in non-canonical space. |
| | If the interrupt vector number points to a descriptor which is not a 64-bit interrupt gate or 64-bit trap gate. |
| | If the descriptor pointed to by the gate selector is outside the descriptor table limit. |
| | If the descriptor pointed to by the gate selector is in non-canonical space. |
| | If the descriptor pointed to by the gate selector is not a code segment. |
| | If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set. |
| | If the descriptor pointed to by the gate selector has DPL > CPL. |
| #SS(0) | If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch. |
| #SS(selector) | If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST). |
| #NP(selector) | If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present. |

| | |
|---|---|
| #TS(selector) | If an attempt to load RSP from the TSS causes an access to non-canonical space. |
| | If the RSP from the TSS is outside descriptor table limits. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

# INVD—Invalidate Internal Caches

| Opcode* | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 08 | INVD | Valid | Valid | Flush internal caches; initiate flushing of external caches. |

**NOTES:**

* See the IA-32 Architecture Compatibility section below.

## Description

Invalidates (flushes) the processor's internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

The INVD instruction is implementation dependent; it may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

## Operation

```
Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution *)
```

## Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | The INVD instruction cannot be executed in virtual-8086 mode. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# INVLPG—Invalidate TLB Entry

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 01/7 | INVLPG *m* | Valid | Valid | Invalidate TLB Entry for page that contains *m.* |

**NOTES:**

* See the IA-32 Architecture Compatibility section below.

## Description

Invalidates (flushes) the translation lookaside buffer (TLB) entry specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes the TLB entry for that page.

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

The INVLPG instruction normally flushes the TLB entry only for the specified page; however, in some cases, it flushes the entire TLB. See “MOV—Move to/from Control Registers” in this chapter for further information on operations that flush the TLB.

This instruction’s operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

## IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

## Operation

```
Flush(RelevantTLBEntries);
Continue; (* Continue execution *)
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| #UD | Operand is a register. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #UD | Operand is a register.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | The INVLPG instruction cannot be executed at the virtual-8086 mode. |

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| #UD | Operand is a register.<br>If the LOCK prefix is used. |

# IRET/IRETD—Interrupt Return

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| CF | IRET | Valid | Valid | Interrupt return (16-bit operand size). |
| CF | IRETD | Valid | Valid | Interrupt return (32-bit operand size). |
| REX.W + CF | IRETQ | Valid | N.E. | Interrupt return (64-bit operand size). |

## Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled “Task Linking” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.W prefix promotes operation to 64 bits (IRETQ). See the summary chart at the beginning of this section for encoding data and limits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE
        IF (IA32_EFER.LMA = 0)
                THEN (* Protected mode *)
                    GOTO PROTECTED-MODE;
                ELSE (* IA-32e mode *)
                    GOTO IA-32e-MODE;
        FI;
FI;
REAL-ADDRESS-MODE;
    IF OperandSize = 32
        THEN
            IF top 12 bytes of stack not within stack limits
                THEN #SS; FI;
            tempEIP ←4 bytes at end of stack
            IF tempEIP[31:16] is not zero THEN #GP(0); FI;
            EIP ←Pop();
            CS ←Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ←Pop();
```

```
            EFLAGS ←(tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
            IF top 6 bytes of stack are not within stack limits
                THEN #SS; FI;
            EIP ←Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS ←Pop(); (* 16-bit pop *)
            EFLAGS[15:0] ←Pop();
    FI;
    END;
PROTECTED-MODE:
    IF VM = 1 (* Virtual-8086 mode: PE = 1, VM = 1 *)
        THEN
            GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE = 1, VM = 1 *)
    FI;
    IF NT = 1
        THEN
            GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
            IF top 12 bytes of stack not within stack limits
                THEN #SS(0); FI;
            tempEIP ←Pop();
            tempCS ←Pop();
            tempEFLAGS ←Pop();
        ELSE (* OperandSize = 16 *)
            IF top 6 bytes of stack are not within stack limits
                                THEN #SS(0); FI;
            tempEIP ←Pop();
            tempCS ←Pop();
            tempEFLAGS ←Pop();
            tempEIP ←tempEIP AND FFFFH;
            tempEFLAGS ←tempEFLAGS AND FFFFH;
    FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN
            GOTO RETURN-TO-VIRTUAL-8086-MODE;
        ELSE
            GOTO PROTECTED-MODE-RETURN;
    FI;
IA-32e-MODE:
    IF NT = 1
        THEN #GP(0);
```

```
    ELSE IF OperandSize = 32
        THEN
            IF top 12 bytes of stack not within stack limits
                THEN #SS(0); FI;
            tempEIP ←Pop();
            tempCS ←Pop();
            tempEFLAGS ←Pop();
        ELSE IF OperandSize = 16
            THEN
                IF top 6 bytes of stack are not within stack limits
                        THEN #SS(0); FI;
                tempEIP ←Pop();
                tempCS ←Pop();
                tempEFLAGS ←Pop();
                tempEIP ←tempEIP AND FFFFH;
                tempEFLAGS ←tempEFLAGS AND FFFFH;
            FI;
        ELSE (* OperandSize = 64 *)
            THEN
                    tempRIP ←Pop();
                    tempCS ←Pop();
                    tempEFLAGS ←Pop();
                    tempRSP ←Pop();
                    tempSS ←Pop();
    FI;
    GOTO IA-32e-MODE-RETURN;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
    IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
        THEN IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                IF instruction pointer not within code segment limits
                    THEN #GP(0); FI;
                EIP ←Pop();
                CS ←Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                EFLAGS ←Pop();
                (* VM, IOPL,VIP and VIF EFLAG bits not modified by pop *)
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack are not within stack limits
                    THEN #SS(0); FI;
                IF instruction pointer not within code segment limits
```

```
                    THEN #GP(0); FI;
                EIP ←Pop();
                EIP ←EIP AND 0000FFFFH;
                CS ←Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ←Pop(); (* IOPL in EFLAGS not modified by pop *)
            FI;
        ELSE
            #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
    IF top 24 bytes of stack are not within stack segment limits
        THEN #SS(0); FI;
    IF instruction pointer not within code segment limits
        THEN #GP(0); FI;
    CS ←tempCS;
    EIP ←tempEIP;
    EFLAGS ←tempEFLAGS;
    TempESP ←Pop();
    TempSS ←Pop();
    ES ←Pop(); (* Pop 2 words; throw away high-order word *)
    DS ←Pop(); (* Pop 2 words; throw away high-order word *)
    FS ←Pop(); (* Pop 2 words; throw away high-order word *)
    GS ←Pop(); (* Pop 2 words; throw away high-order word *)
    SS:ESP ←TempSS:TempESP;
    CPL ←3;
    (* Resume execution in Virtual-8086 mode *)
END;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    Read segment selector in link field of current TSS;
    IF local/global bit is set to local
    or index not within GDT limits
        THEN #TS (TSS selector); FI;
    Access TSS for task specified in link field of current TSS;
    IF TSS descriptor type is not TSS or if the TSS is marked not busy
        THEN #TS (TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
```

```
    IF EIP is not within code segment limit
        THEN #GP(0); FI;
END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
    IF return code segment selector is NULL
        THEN GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
        THEN GP(selector); FI;
    Read segment descriptor pointed to by the return code segment selector;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
    IF new mode ≠ 64-Bit Mode
        THEN
            IF tempEIP is not within code segment limits
                THEN #GP(0); FI;
            EIP ←tempEIP;
        ELSE (* new mode = 64-bit mode *)
            IF tempRIP is non-canonical
                    THEN #GP(0); FI;
            RIP ←tempRIP;
    FI;
    CS ←tempCS; (* Segment descriptor information also loaded *)
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ←tempEFLAGS;
    IF OperandSize = 32 or OperandSize = 64
        THEN EFLAGS(RF, AC, ID) ←tempEFLAGS; FI;
    IF CPL ≤IOPL
        THEN EFLAGS(IF) ←tempEFLAGS; FI;
    IF CPL = 0
         THEN (* VM = 0 in flags image *)
```

```
            EFLAGS(IOPL) ←tempEFLAGS;
            IF OperandSize = 32 or OperandSize = 64
                THEN EFLAGS(VIF, VIP) ←tempEFLAGS; FI;
        FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF OperandSize = 32
        THEN
            IF top 8 bytes on stack are not within limits
                THEN #SS(0); FI;
        ELSE (* OperandSize = 16 *)
            IF top 4 bytes on stack are not within limits
                THEN #SS(0); FI;
    FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(SSselector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    or the stack segment descriptor does not indicate a a writable data segment;
    or the stack segment DPL ≠ RPL of the return code segment selector
        THEN #GP(SS selector); FI;
    IF stack segment is not present
        THEN #SS(SS selector); FI;
    IF new mode ≠ 64-Bit Mode
        THEN
            IF tempEIP is not within code segment limits
                THEN #GP(0); FI;
            EIP ←tempEIP;
        ELSE (* new mode = 64-bit mode *)
            IF tempRIP is non-canonical
                    THEN #GP(0); FI;
            RIP ←tempRIP;
    FI;
    CS ←tempCS;
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ←tempEFLAGS;
    IF OperandSize = 32
        THEN EFLAGS(RF, AC, ID) ←tempEFLAGS; FI;
    IF CPL ≤IOPL
        THEN EFLAGS(IF) ←tempEFLAGS; FI;
```

```
    IF CPL = 0
        THEN
            EFLAGS(IOPL) ←tempEFLAGS;
            IF OperandSize = 32
                THEN EFLAGS(VM, VIF, VIP) ←tempEFLAGS; FI;
            IF OperandSize = 64
                THEN EFLAGS(VIF, VIP) ←tempEFLAGS; FI;
    FI;
    CPL ←RPL of the return code segment selector;
    FOR each of segment register (ES, FS, GS, and DS)
        DO
            IF segment register points to data or non-conforming code segment
            and CPL > segment descriptor DPL (* Stored in hidden part of segment register *)
                THEN (* Segment register invalid *)
                    SegmentSelector ←0; (* NULL segment selector *)
            FI;
        OD;
END;

IA-32e-MODE-RETURN: (* IA32_EFER.LMA = 1, PE = 1 *)
    IF ( (return code segment selector is NULL) or (return RIP is non-canonical) or
            (SS selector is NULL going back to compatibility mode) or
            (SS selector is NULL going back to CPL3 64-bit mode) or
            (RPL <> CPL going back to non-CPL3 64-bit mode for a NULL SS selector) )
        THEN GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
        THEN GP(selector); FI;
    Read segment descriptor pointed to by the return code segment selector;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;
```

## Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the return code or stack segment selector is NULL. |
| | If the return instruction pointer is not within the return code segment limit. |
| #GP(selector) | If a segment selector index is outside its descriptor table limits. |
| | If the return code segment selector RPL is greater than the CPL. |
| | If the DPL of a conforming-code segment is greater than the return code segment selector RPL. |
| | If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector. |
| | If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector. |
| | If the stack segment is not a writable data segment. |
| | If the stack segment selector RPL is not equal to the RPL of the return code segment selector. |
| | If the segment descriptor for a code segment does not indicate it is a code segment. |
| | If the segment selector for a TSS has its local/global bit set for local. |
| | If a TSS segment descriptor specifies that the TSS is not busy. |
| | If a TSS segment descriptor specifies that the TSS is not available. |
| #SS(0) | If the top bytes of stack are not within stack limits. |
| #NP(selector) | If the return code or stack segment is not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If the return instruction pointer is not within the return code segment limit. |
| #SS | If the top bytes of stack are not within stack limits. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If the return instruction pointer is not within the return code segment limit. |
| | IF IOPL not equal to 3. |
| #PF(fault-code) | If a page fault occurs. |
| #SS(0) | If the top bytes of stack are not within stack limits. |
| #AC(0) | If an unaligned memory reference occurs and alignment checking is enabled. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

| | |
|---|---|
| #GP(0) | If EFLAGS.NT[bit 14] = 1. |

Other exceptions same as in Protected Mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If EFLAGS.NT[bit 14] = 1. |
| | If the return code segment selector is NULL. |
| | If the stack segment selector is NULL going back to compatibility mode. |
| | If the stack segment selector is NULL going back to CPL3 64-bit mode. |
| | If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode. |
| | If the return instruction pointer is not within the return code segment limit. |
| | If the return instruction pointer is non-canonical. |
| #GP(Selector) | If a segment selector index is outside its descriptor table limits. |
| | If a segment descriptor memory address is non-canonical. |
| | If the segment descriptor for a code segment does not indicate it is a code segment. |
| | If the proposed new code segment descriptor has both the D-bit and L-bit set. |
| | If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector. |
| | If CPL is greater than the RPL of the code segment selector. |
| | If the DPL of a conforming-code segment is greater than the return code segment selector RPL. |
| | If the stack segment is not a writable data segment. |
| | If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector. |

| | |
|---|---|
| | If the stack segment selector RPL is not equal to the RPL of the return code segment selector. |
| #SS(0) | If an attempt to pop a value off the stack violates the SS limit. |
| | If an attempt to pop a value off the stack causes a non-canonical address to be referenced. |
| #NP(selector) | If the return code or stack segment is not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled. |
| #UD | If the LOCK prefix is used. |

## Jcc—Jump if Condition Is Met

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 77 *cb* | JA *rel8* | Valid | Valid | Jump short if above (CF=0 and ZF=0). |
| 73 *cb* | JAE *rel8* | Valid | Valid | Jump short if above or equal (CF=0). |
| 72 *cb* | JB *rel8* | Valid | Valid | Jump short if below (CF=1). |
| 76 *cb* | JBE *rel8* | Valid | Valid | Jump short if below or equal (CF=1 or ZF=1). |
| 72 *cb* | JC *rel8* | Valid | Valid | Jump short if carry (CF=1). |
| E3 *cb* | JCXZ *rel8* | N.E. | Valid | Jump short if CX register is 0. |
| E3 *cb* | JECXZ *rel8* | Valid | Valid | Jump short if ECX register is 0. |
| E3 *cb* | JRCXZ *rel8* | Valid | N.E. | Jump short if RCX register is 0. |
| 74 *cb* | JE *rel8* | Valid | Valid | Jump short if equal (ZF=1). |
| 7F *cb* | JG *rel8* | Valid | Valid | Jump short if greater (ZF=0 and SF=OF). |
| 7D *cb* | JGE *rel8* | Valid | Valid | Jump short if greater or equal (SF=OF). |
| 7C *cb* | JL *rel8* | Valid | Valid | Jump short if less (SF≠ OF). |
| 7E *cb* | JLE *rel8* | Valid | Valid | Jump short if less or equal (ZF=1 or SF≠ OF). |
| 76 *cb* | JNA *rel8* | Valid | Valid | Jump short if not above (CF=1 or ZF=1). |
| 72 *cb* | JNAE *rel8* | Valid | Valid | Jump short if not above or equal (CF=1). |
| 73 *cb* | JNB *rel8* | Valid | Valid | Jump short if not below (CF=0). |
| 77 *cb* | JNBE *rel8* | Valid | Valid | Jump short if not below or equal (CF=0 and ZF=0). |
| 73 *cb* | JNC *rel8* | Valid | Valid | Jump short if not carry (CF=0). |
| 75 *cb* | JNE *rel8* | Valid | Valid | Jump short if not equal (ZF=0). |
| 7E *cb* | JNG *rel8* | Valid | Valid | Jump short if not greater (ZF=1 or SF≠ OF). |
| 7C *cb* | JNGE *rel8* | Valid | Valid | Jump short if not greater or equal (SF≠ OF). |
| 7D *cb* | JNL *rel8* | Valid | Valid | Jump short if not less (SF=OF). |
| 7F *cb* | JNLE *rel8* | Valid | Valid | Jump short if not less or equal (ZF=0 and SF=OF). |
| 71 *cb* | JNO *rel8* | Valid | Valid | Jump short if not overflow (OF=0). |
| 7B *cb* | JNP *rel8* | Valid | Valid | Jump short if not parity (PF=0). |
| 79 *cb* | JNS *rel8* | Valid | Valid | Jump short if not sign (SF=0). |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 75 *cb* | JNZ *rel8* | Valid | Valid | Jump short if not zero (ZF=0). |
| 70 *cb* | JO *rel8* | Valid | Valid | Jump short if overflow (OF=1). |
| 7A *cb* | JP *rel8* | Valid | Valid | Jump short if parity (PF=1). |
| 7A *cb* | JPE *rel8* | Valid | Valid | Jump short if parity even (PF=1). |
| 7B *cb* | JPO *rel8* | Valid | Valid | Jump short if parity odd (PF=0). |
| 78 *cb* | JS *rel8* | Valid | Valid | Jump short if sign (SF=1). |
| 74 *cb* | JZ *rel8* | Valid | Valid | Jump short if zero (ZF ←1). |
| 0F 87 *cw* | JA *rel16* | N.S. | Valid | Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode. |
| 0F 87 *cd* | JA *rel32* | Valid | Valid | Jump near if above (CF=0 and ZF=0). |
| 0F 83 *cw* | JAE *rel16* | N.S. | Valid | Jump near if above or equal (CF=0). Not supported in 64-bit mode. |
| 0F 83 *cd* | JAE *rel32* | Valid | Valid | Jump near if above or equal (CF=0). |
| 0F 82 *cw* | JB *rel16* | N.S. | Valid | Jump near if below (CF=1). Not supported in 64-bit mode. |
| 0F 82 *cd* | JB *rel32* | Valid | Valid | Jump near if below (CF=1). |
| 0F 86 *cw* | JBE *rel16* | N.S. | Valid | Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode. |
| 0F 86 *cd* | JBE *rel32* | Valid | Valid | Jump near if below or equal (CF=1 or ZF=1). |
| 0F 82 *cw* | JC *rel16* | N.S. | Valid | Jump near if carry (CF=1). Not supported in 64-bit mode. |
| 0F 82 *cd* | JC *rel32* | Valid | Valid | Jump near if carry (CF=1). |
| 0F 84 *cw* | JE *rel16* | N.S. | Valid | Jump near if equal (ZF=1). Not supported in 64-bit mode. |
| 0F 84 *cd* | JE *rel32* | Valid | Valid | Jump near if equal (ZF=1). |
| 0F 84 *cw* | JZ *rel16* | N.S. | Valid | Jump near if 0 (ZF=1). Not supported in 64-bit mode. |
| 0F 84 *cd* | JZ *rel32* | Valid | Valid | Jump near if 0 (ZF=1). |
| 0F 8F *cw* | JG *rel16* | N.S. | Valid | Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode. |
| 0F 8F *cd* | JG *rel32* | Valid | Valid | Jump near if greater (ZF=0 and SF=OF). |
| 0F 8D *cw* | JGE *rel16* | N.S. | Valid | Jump near if greater or equal (SF=OF). Not supported in 64-bit mode. |
| 0F 8D *cd* | JGE *rel32* | Valid | Valid | Jump near if greater or equal (SF=OF). |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 8C *cw* | JL *rel16* | N.S. | Valid | Jump near if less (SF≠ OF). Not supported in 64-bit mode. |
| 0F 8C *cd* | JL *rel32* | Valid | Valid | Jump near if less (SF≠ OF). |
| 0F 8E *cw* | JLE *rel16* | N.S. | Valid | Jump near if less or equal (ZF=1 or SF≠ OF). Not supported in 64-bit mode. |
| 0F 8E *cd* | JLE *rel32* | Valid | Valid | Jump near if less or equal (ZF=1 or SF≠ OF). |
| 0F 86 *cw* | JNA *rel16* | N.S. | Valid | Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode. |
| 0F 86 *cd* | JNA *rel32* | Valid | Valid | Jump near if not above (CF=1 or ZF=1). |
| 0F 82 *cw* | JNAE *rel16* | N.S. | Valid | Jump near if not above or equal (CF=1). Not supported in 64-bit mode. |
| 0F 82 *cd* | JNAE *rel32* | Valid | Valid | Jump near if not above or equal (CF=1). |
| 0F 83 *cw* | JNB *rel16* | N.S. | Valid | Jump near if not below (CF=0). Not supported in 64-bit mode. |
| 0F 83 *cd* | JNB *rel32* | Valid | Valid | Jump near if not below (CF=0). |
| 0F 87 *cw* | JNBE *rel16* | N.S. | Valid | Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode. |
| 0F 87 *cd* | JNBE *rel32* | Valid | Valid | Jump near if not below or equal (CF=0 and ZF=0). |
| 0F 83 *cw* | JNC *rel16* | N.S. | Valid | Jump near if not carry (CF=0). Not supported in 64-bit mode. |
| 0F 83 *cd* | JNC *rel32* | Valid | Valid | Jump near if not carry (CF=0). |
| 0F 85 *cw* | JNE *rel16* | N.S. | Valid | Jump near if not equal (ZF=0). Not supported in 64-bit mode. |
| 0F 85 *cd* | JNE *rel32* | Valid | Valid | Jump near if not equal (ZF=0). |
| 0F 8E *cw* | JNG *rel16* | N.S. | Valid | Jump near if not greater (ZF=1 or SF≠ OF). Not supported in 64-bit mode. |
| 0F 8E *cd* | JNG *rel32* | Valid | Valid | Jump near if not greater (ZF=1 or SF≠ OF). |
| 0F 8C *cw* | JNGE *rel16* | N.S. | Valid | Jump near if not greater or equal (SF≠ OF). Not supported in 64-bit mode. |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 8C *cd* | JNGE *rel32* | Valid | Valid | Jump near if not greater or equal (SF≠ OF). |
| 0F 8D *cw* | JNL *rel16* | N.S. | Valid | Jump near if not less (SF=OF). Not supported in 64-bit mode. |
| 0F 8D *cd* | JNL *rel32* | Valid | Valid | Jump near if not less (SF=OF). |
| 0F 8F *cw* | JNLE *rel16* | N.S. | Valid | Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode. |
| 0F 8F *cd* | JNLE *rel32* | Valid | Valid | Jump near if not less or equal (ZF=0 and SF=OF). |
| 0F 81 *cw* | JNO *rel16* | N.S. | Valid | Jump near if not overflow (OF=0). Not supported in 64-bit mode. |
| 0F 81 *cd* | JNO *rel32* | Valid | Valid | Jump near if not overflow (OF=0). |
| 0F 8B *cw* | JNP *rel16* | N.S. | Valid | Jump near if not parity (PF=0). Not supported in 64-bit mode. |
| 0F 8B *cd* | JNP *rel32* | Valid | Valid | Jump near if not parity (PF=0). |
| 0F 89 *cw* | JNS *rel16* | N.S. | Valid | Jump near if not sign (SF=0). Not supported in 64-bit mode. |
| 0F 89 *cd* | JNS *rel32* | Valid | Valid | Jump near if not sign (SF=0). |
| 0F 85 *cw* | JNZ *rel16* | N.S. | Valid | Jump near if not zero (ZF=0). Not supported in 64-bit mode. |
| 0F 85 *cd* | JNZ *rel32* | Valid | Valid | Jump near if not zero (ZF=0). |
| 0F 80 *cw* | JO *rel16* | N.S. | Valid | Jump near if overflow (OF=1). Not supported in 64-bit mode. |
| 0F 80 *cd* | JO *rel32* | Valid | Valid | Jump near if overflow (OF=1). |
| 0F 8A *cw* | JP *rel16* | N.S. | Valid | Jump near if parity (PF=1). Not supported in 64-bit mode. |
| 0F 8A *cd* | JP *rel32* | Valid | Valid | Jump near if parity (PF=1). |
| 0F 8A *cw* | JPE *rel16* | N.S. | Valid | Jump near if parity even (PF=1). Not supported in 64-bit mode. |
| 0F 8A *cd* | JPE *rel32* | Valid | Valid | Jump near if parity even (PF=1). |
| 0F 8B *cw* | JPO *rel16* | N.S. | Valid | Jump near if parity odd (PF=0). Not supported in 64-bit mode. |
| 0F 8B *cd* | JPO *rel32* | Valid | Valid | Jump near if parity odd (PF=0). |
| 0F 88 *cw* | JS *rel16* | N.S. | Valid | Jump near if sign (SF=1). Not supported in 64-bit mode. |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 88 *cd* | JS *rel32* | Valid | Valid | Jump near if sign (SF=1). |
| 0F 84 *cw* | JZ *rel16* | N.S. | Valid | Jump near if 0 (ZF=1). Not supported in 64-bit mode. |
| 0F 84 *cd* | JZ *rel32* | Valid | Valid | Jump near if 0 (ZF=1). |

## Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the J*cc* instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16,* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of –128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each J*cc* mnemonic are given in the "Description" column of the table on the preceding page. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The J*cc* instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the J*cc* instruction, and then access the target with an unconditional far jump (JMP instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The JRCXZ, JECXZ and JCXZ instructions differ from other J*cc* instructions because they do not check status flags. Instead, they check RCX, ECX or CX for 0. The register

checked is determined by the address-size attribute. These instructions are useful when used at the beginning of a loop that terminates with a conditional loop instruction (such as LOOPNE). They can be used to prevent an instruction sequence from entering a loop when RCX, ECX or CX is 0. This would cause the loop to execute $2^{64}$, $2^{32}$ or 64K times (not zero times).

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

In 64-bit mode, operand size is fixed at 64 bits. JMP Short is RIP = RIP + 8-bit offset sign extended to 64 bits. JMP Near is RIP = RIP + 32-bit offset sign extended to 64-bits.

## Operation

```
IF condition
    THEN
        tempEIP ←EIP + SignExtend(DEST);
        IF OperandSize = 16
            THEN tempEIP ←tempEIP AND 0000FFFFH;
        FI;
    IF tempEIP is not within code segment limit
        THEN #GP(0);
        ELSE EIP ←tempEIP
    FI;
FI;
```

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the offset being jumped to is beyond the limits of the CS segment. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the memory address is in a non-canonical form. |
| #UD | If the LOCK prefix is used. |

# JMP—Jump

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| EB *cb* | JMP *rel8* | Valid | Valid | Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits |
| E9 *cw* | JMP *rel16* | N.S. | Valid | Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode. |
| E9 *cd* | JMP *rel32* | Valid | Valid | Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits |
| FF /4 | JMP *r/m16* | N.S. | Valid | Jump near, absolute indirect, address = sign-extended *r/m16.* Not supported in 64-bit mode. |
| FF /4 | JMP *r/m32* | N.S. | Valid | Jump near, absolute indirect, address = sign-extended *r/m32.* Not supported in 64-bit mode. |
| FF /4 | JMP *r/m64* | Valid | N.E. | Jump near, absolute indirect, RIP = 64-Bit offset from register or memory |
| EA *cd* | JMP *ptr16:16* | Inv. | Valid | Jump far, absolute, address given in operand |
| EA *cp* | JMP *ptr16:32* | Inv. | Valid | Jump far, absolute, address given in operand |
| FF /5 | JMP *m16:16* | Valid | Valid | Jump far, absolute indirect, address given in *m16:16* |
| FF /5 | JMP *m16:32* | Valid | Valid | Jump far, absolute indirect, address given in *m16:32.* |
| REX.W + FF /5 | JMP *m16:64* | Valid | N.E. | Jump far, absolute indirect, address given in *m16:64.* |

## Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.

- Short jump—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 6, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the JMP instruction).

**Near and Short Jumps.** When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (*rel8, rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

**Far Jumps in Real-Address or Virtual-8086 Mode.** When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

**Far Jumps in Protected Mode.** When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.

- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 6 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for detailed information on the mechanics of a task switch.

Note that when you execute at task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag

and save the previous task link information, allowing a return to the calling task with an IRET instruction.

**In 64-Bit Mode** — The instruction's operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF near jump
    IF 64-bit Mode
        THEN
            IF near relative jump
             THEN
                tempRIP ←RIP +DEST; (* RIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempRIP ←DEST;
            FI:
        ELSE
            IF near relative jump
             THEN
                tempEIP ←EIP +DEST; (* EIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempEIP ←DEST;
            FI:
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
    and tempEIP outside code segment limit
        THEN #GP(0); FI
    IF 64-bit mode and tempRIP is not canonical
        THEN #GP(0);
    FI;
    IF OperandSize = 32
         THEN
            EIP ←tempEIP;
         ELSE
            IF OperandSize = 16
                THEN (* OperandSize = 16 *)
                    EIP ←tempEIP AND 0000FFFFH;
                ELSE (* OperandSize = 64)
                    RIP ←tempRIP;
            FI;
     FI;
```

```
FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        tempEIP ←DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
        IF tempEIP is beyond code segment limit
            THEN #GP(0); FI;
        CS ←DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
        IF OperandSize = 32
            THEN
                EIP ←tempEIP; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                EIP ←tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
        FI;
FI;
IF far jump and (PE = 1 and VM = 0)
(* IA-32e mode or protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
        or segment selector in target operand NULL
                THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector); FI;
        Read type and access rights of segment descriptor;
        IF (EFER.LMA = 0)
            THEN
                IF segment type is not a conforming or nonconforming code
                segment, call gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment
                call gate
                    THEN #GP(segment selector); FI;
        FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
    ELSE
        #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
```

```
            THEN GP(new code segment selector); FI;
        IF DPL > CPL
            THEN #GP(segment selector); FI;
        IF segment not present
            THEN #NP(segment selector); FI;
        tempEIP ←DEST(Offset);
        IF OperandSize = 16
            THEN tempEIP ←tempEIP AND 0000FFFFH;
        FI;
        IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and
        tempEIP outside code segment limit
            THEN #GP(0); FI
        IF tempEIP is non-canonical
            THEN #GP(0); FI;
        CS ←DEST[segment selector]; (* Segment descriptor information also loaded *)
        CS(RPL) ←CPL
        EIP ←tempEIP;
END;
NONCONFORMING-CODE-SEGMENT:
        IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
            THEN GP(new code segment selector); FI;
        IF (RPL > CPL) OR (DPL ≠ CPL)
            THEN #GP(code segment selector); FI;
        IF segment not present
            THEN #NP(segment selector); FI;
        tempEIP ←DEST(Offset);
        IF OperandSize = 16
            THEN tempEIP ←tempEIP AND 0000FFFFH; FI;
        IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
        and tempEIP outside code segment limit
            THEN #GP(0); FI
        IF tempEIP is non-canonical THEN #GP(0); FI;
        CS ←DEST[segment selector]; (* Segment descriptor information also loaded *)
        CS(RPL) ←CPL;
        EIP ←tempEIP;
END;

CALL-GATE:
        IF call gate DPL < CPL
        or call gate DPL < call gate segment-selector RPL
                THEN #GP(call gate selector); FI;
        IF call gate not present
            THEN #NP(call gate selector); FI;
        IF call gate code-segment selector is NULL
```

```
        THEN #GP(0); FI;
    IF call gate code-segment selector index outside descriptor table limits
        THEN #GP(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
    or code-segment segment descriptor is conforming and DPL > CPL
    or code-segment segment descriptor is non-conforming and DPL ≠ CPL
            THEN #GP(code segment selector); FI;
    IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
    or code-segment segment descriptor has both L-Bit and D-bit set)
            THEN #GP(code segment selector); FI;
    IF code segment is not present
        THEN #NP(code-segment selector); FI;
     IF instruction pointer is not within code-segment limit
        THEN #GP(0); FI;
     tempEIP ←DEST(Offset);
     IF GateSize = 16
         THEN tempEIP ←tempEIP AND 0000FFFFH; FI;
    IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
    outside code segment limit
        THEN #GP(0); FI
    CS ←DEST[SegmentSelector); (* Segment descriptor information also loaded *)
    CS(RPL) ←CPL;
    EIP ←tempEIP;
END;
TASK-GATE:
    IF task gate DPL < CPL
    or task gate DPL < task gate segment-selector RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
    or TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
     IF TSS not present
        THEN #NP(TSS selector); FI;
     SWITCH-TASKS to TSS;
     IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
TASK-STATE-SEGMENT:
    IF TSS DPL < CPL
```

```
    or TSS DPL < TSS segment-selector RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
```

## Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If offset in target operand, call gate, or TSS is beyond the code segment limits. |
| | If the segment selector in the destination operand, call gate, task gate, or TSS is NULL. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #GP(selector) | If the segment selector index is outside descriptor table limits. |
| | If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment. |
| | If the DPL for a nonconforming-code segment is not equal to the CPL |
| | (When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL. |
| | If the DPL for a conforming-code segment is greater than the CPL. |
| | If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector. |
| | If the segment descriptor for selector in a call gate does not indicate it is a code segment. |
| | If the segment descriptor for the segment selector in a task gate does not indicate an available TSS. |

| | |
|---|---|
| | If the segment selector for a TSS has its local/global bit set for local. |
| | If a TSS segment descriptor specifies that the TSS is busy or not available. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NP (selector) | If the code segment being accessed is not present. |
| | If call gate, task gate, or TSS not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.) |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If the target operand is beyond the code segment limits. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. (Only occurs when fetching target from memory.) |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory address is non-canonical. |

| | |
|---|---|
| | If target offset in destination operand is non-canonical. |
| | If target offset in destination operand is beyond the new code segment limit. |
| | If the segment selector in the destination operand is NULL. |
| | If the code segment selector in the 64-bit gate is NULL. |
| #GP(selector) | If the code segment or 64-bit call gate is outside descriptor table limits. |
| | If the code segment or 64-bit call gate overlaps non-canonical space. |
| | If the segment descriptor from a 64-bit call gate is in non-canonical space. |
| | If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate. |
| | If the segment descriptor pointed to by the segment selector in the destination operand is a code segment, and has both the D-bit and the L-bit set. |
| | If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL. |
| | If the DPL for a conforming-code segment is greater than the CPL. |
| | If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate. |
| | If the upper type field of a 64-bit call gate is not 0x0. |
| | If the segment selector from a 64-bit call gate is beyond the descriptor table limits. |
| | If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear. |
| | If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment. |
| | If the code segment is non-confirming and CPL ≠ DPL. |
| | If the code segment is confirming and CPL < DPL. |
| #NP(selector) | If a code segment or 64-bit call gate is not present. |
| #UD | (64-bit mode only) If a far jump is direct to an absolute address in memory. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# LAHF—Load Status Flags into AH Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 9F | LAHF | Invalid* | Valid | Load: AH ←EFLAGS(SF:ZF:0:AF:0:PF:1:CF). |

**NOTES:**

* Valid in specific steppings. See Description section.

## Description

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

## Operation

```
IF 64-Bit Mode
    THEN
        IF CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1;
            THEN AH ←RFLAGS(SF:ZF:0:AF:0:PF:1:CF);
            ELSE #UD;
        FI;
    ELSE
        AH ←EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
FI;
```

## Flags Affected

None. The state of the flags in the EFLAGS register is not affected.

## Protected Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #UD | If CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 0.<br>If the LOCK prefix is used. |

# LAR—Load Access Rights Byte

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 02 /r | LAR *r16, r16/m16* | Valid | Valid | *r16* ←*r16/m16* masked by FF00H. |
| 0F 02 /r | LAR *r32, r32/m16*[1] | Valid | Valid | *r32* ←*r32/m16* masked by 00FxFF00H |
| REX.W + 0F 02 /r | LAR *r64, r32/m16*[1] | Valid | N.E. | *r64* ←*r32/m16* masked by 00FxFF00H and zero extended |

**NOTES:**

1. For all loads (regardless of source or destination sizing) only bits 16-0 are used. Other bits are ignored.

## Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the flag register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

When the operand size is 32 bits, the access rights for a segment descriptor include the type and DPL fields and the S, P, AVL, D/B, and G flags, all of which are located in the second doubleword (bytes 4 through 7) of the segment descriptor. The doubleword is masked by 00FXFF00H before it is loaded into the destination operand. When the operand size is 16 bits, the access rights include the type and DPL fields. Here, the two lower-order bytes of the doubleword are masked by FF00H before being loaded into the destination operand.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-57.

- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode and IA-32e mode.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.W prefix permits access to 64-bit registers as destination.

When the destination operand size is 64 bits, the access rights are loaded from the second doubleword (bytes 4 through 7) of the segment descriptor. The doubleword is masked by 00FXFF00H and zero extended to 64 bits before it is loaded into the destination operand.

**Table 3-57. Segment and Gate Types**

| Type | Protected Mode | | IA-32e Mode | |
|---|---|---|---|---|
| | Name | Valid | Name | Valid |
| 0 | Reserved | No | Reserved | No |
| 1 | Available 16-bit TSS | Yes | Reserved | No |
| 2 | LDT | Yes | LDT | No |
| 3 | Busy 16-bit TSS | Yes | Reserved | No |
| 4 | 16-bit call gate | Yes | Reserved | No |
| 5 | 16-bit/32-bit task gate | Yes | Reserved | No |
| 6 | 16-bit interrupt gate | No | Reserved | No |
| 7 | 16-bit trap gate | No | Reserved | No |
| 8 | Reserved | No | Reserved | No |
| 9 | Available 32-bit TSS | Yes | Available 64-bit TSS | Yes |
| A | Reserved | No | Reserved | No |
| B | Busy 32-bit TSS | Yes | Busy 64-bit TSS | Yes |
| C | 32-bit call gate | Yes | 64-bit call gate | Yes |
| D | Reserved | No | Reserved | No |
| E | 32-bit interrupt gate | No | 64-bit interrupt gate | No |
| F | 32-bit trap gate | No | 64-bit trap gate | No |

## Operation

```
IF Offset(SRC) > descriptor table limit
    THEN
```

```
        ZF = 0;
    ELSE

        IF SegmentDescriptor(Type) ≠ conforming code segment
        and (CPL > DPL) or (RPL > DPL)
        or segment type is not valid for instruction
            THEN
                ZF ←0
            ELSE

                TEMP ←Read segment descriptor ;
                IF OperandSize = 64
                    THEN
                        DEST ←(ACCESSRIGHTWORD(TEMP) AND 00000000_00FxFF00H);
                    ELSE (* OperandSize = 32*)
                        DEST ←(ACCESSRIGHTWORD(TEMP) AND 00FxFF00H);
                    ELSE (* OperandSize = 16 *)
                        DEST ←(ACCESSRIGHTWORD(TEMP) AND FF00H);
                FI;
        FI;
FI:
```

## Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is set to 0.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | The LAR instruction is not recognized in real-address mode. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | The LAR instruction cannot be executed in virtual-8086 mode. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If the memory operand effective address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory operand effective address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# LDDQU—Load Unaligned Integer 128 Bits

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F F0 /r | LDDQU *xmm1, mem* | Valid | Valid | Load unaligned data from *mem* and return double quadword in *xmm1*. |

## Description

The instruction is *functionally similar* to MOVDQU xmm, m128 for loading from memory. That is: 16 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 16-byte boundary. Up to 32 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to MOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by LDDQU be modified and stored to the same location, use MOVDQU or MOVDQA instead of LDDQU. To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the MOVDQA instruction.

## Implementation Notes

- If the source is aligned to a 16-byte boundary, based on the implementation, the 16 bytes may be loaded more than once. For that reason, the usage of LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using MOVDQU.
- This instruction is a replacement for MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use MOVDQA store-load pairs when data is 128-bit aligned or MOVDQU store-load pairs when data is 128-bit unaligned.
- If the memory address is not aligned on 16-byte boundary, some implementations may load up to 32 bytes and return 16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
xmm[127:0] = m128;
```

## Intel C/C++ Compiler Intrinsic Equivalent

LDDQU __m128i _mm_lddqu_si128(__m128i const *p)

## Numeric Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR4.OSFXSR[bit 9] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

## Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | If a page fault occurs. |

# LDMXCSR—Load MXCSR Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F,AE,/2 | LDMXCSR *m32* | Valid | Valid | Load MXCSR register from *m32*. |

## Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See "MXCSR Control and Status Register" in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the MXCSR register and its contents.

The LDMXCSR instruction is typically used in conjunction with the STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 1F80H.

If a LDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next SSE or SSE2 instruction that causes that particular SIMD floating-point exception to be reported.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
MXCSR ←m32;
```

## C/C++ Compiler Intrinsic Equivalent

_mm_setcsr(unsigned int i)

## Numeric Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS, or GS segments. |
| | For an attempt to set reserved bits in MXCSR. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |

| | |
|---|---|
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to FFFFH. |
| | For an attempt to set reserved bits in MXCSR. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | For an attempt to set reserved bits in MXCSR. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# LDS/LES/LFS/LGS/LSS—Load Far Pointer

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| C5 /*r* | LDS *r16,m16:16* | Invalid | Valid | Load DS:*r16* with far pointer from memory. |
| C5 /*r* | LDS *r32,m16:32* | Invalid | Valid | Load DS:*r32* with far pointer from memory. |
| 0F B2 /*r* | LSS *r16,m16:16* | Valid | Valid | Load SS:*r16* with far pointer from memory. |
| 0F B2 /*r* | LSS *r32,m16:32* | Valid | Valid | Load SS:*r32* with far pointer from memory. |
| REX + 0F B2 /*r* | LSS *r64,m16:64* | Valid | N.E. | Load SS:*r64* with far pointer from memory. |
| C4 /*r* | LES *r16,m16:16* | Invalid | Valid | Load ES:*r16* with far pointer from memory. |
| C4 /*r* | LES *r32,m16:32* | Invalid | Valid | Load ES:*r32* with far pointer from memory. |
| 0F B4 /*r* | LFS *r16,m16:16* | Valid | Valid | Load FS:*r16* with far pointer from memory. |
| 0F B4 /*r* | LFS *r32,m16:32* | Valid | Valid | Load FS:*r32* with far pointer from memory. |
| REX + 0F B4 /*r* | LFS *r64,m16:64* | Valid | N.E. | Load FS:*r64* with far pointer from memory. |
| 0F B5 /*r* | LGS *r16,m16:16* | Valid | Valid | Load GS:*r16* with far pointer from memory. |
| 0F B5 /*r* | LGS *r32,m16:32* | Valid | Valid | Load GS:*r32* with far pointer from memory. |
| REX + 0F B5 /*r* | LGS *r64,m16:64* | Valid | N.E. | Load GS:*r64* with far pointer from memory. |

## Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a NULL selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a NULL selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.W promotes operation to specify a source operand referencing an 80-bit pointer (16-bit selector, 64-bit offset) in memory. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
64-BIT_MODE
    IF SS is loaded
        THEN
            IF SegmentSelector = NULL and ( (RPL = 3) or
                    (RPL ≠ 3 and RPL ≠ CPL) )
                THEN #GP(0);
            ELSE IF descriptor is in non-canonical space
                THEN #GP(0); FI;
            ELSE IF Segment selector index is not within descriptor table limits
                    or segment selector RPL ≠ CPL
                    or access rights indicate nonwritable data segment
                    or DPL ≠ CPL
                THEN #GP(selector); FI;
            ELSE IF Segment marked not present
                THEN #SS(selector); FI;
            FI;
            SS ←SegmentSelector(SRC);
            SS ←SegmentDescriptor([SRC]);
    ELSE IF attempt to load DS, or ES
        THEN #UD;
    ELSE IF FS, or GS is loaded with non-NULL segment selector
        THEN IF Segment selector index is not within descriptor table limits
            or access rights indicate segment neither data nor readable code segment
            or segment is data or nonconforming-code segment
            and ( RPL > DPL or CPL > DPL)
                THEN #GP(selector); FI;
            ELSE IF Segment marked not present
                THEN #NP(selector); FI;
```

```
            FI;
            SegmentRegister ←SegmentSelector(SRC) ;
            SegmentRegister ←SegmentDescriptor([SRC]);
        FI;
    ELSE IF FS, or GS is loaded with a NULL selector:
        THEN
            SegmentRegister ←NULLSelector;
            SegmentRegister(DescriptorValidBit) ←0; FI; (* Hidden flag;
                not accessible by software *)
    FI;
    DEST ←Offset(SRC);

PREOTECTED MODE OR COMPATIBILITY MODE;
    IF SS is loaded
        THEN
            IF SegementSelector = NULL
                THEN #GP(0);
            ELSE IF Segment selector index is not within descriptor table limits
                        or segment selector RPL ≠ CPL
                        or access rights indicate nonwritable data segment
                        or DPL ≠ CPL
                THEN #GP(selector); FI;
            ELSE IF Segment marked not present
                THEN #SS(selector); FI;
            FI;
            SS ←SegmentSelector(SRC);
            SS ←SegmentDescriptor([SRC]);
    ELSE IF DS, ES, FS, or GS is loaded with non-NULL segment selector
        THEN IF Segment selector index is not within descriptor table limits
            or access rights indicate segment neither data nor readable code segment
            or segment is data or nonconforming-code segment
            and (RPL > DPL or CPL > DPL)
                THEN #GP(selector); FI;
            ELSE IF Segment marked not present
                THEN #NP(selector); FI;
            FI;
            SegmentRegister ←SegmentSelector(SRC) AND RPL;
            SegmentRegister ←SegmentDescriptor([SRC]);
        FI;
    ELSE IF DS, ES, FS, or GS is loaded with a NULL selector:
        THEN
            SegmentRegister ←NULLSelector;
            SegmentRegister(DescriptorValidBit) ←0; FI; (* Hidden flag;
                not accessible by software *)
```

```
    FI;
    DEST ←Offset(SRC);

Real-Address or Virtual-8086 Mode
    SegmentRegister ←SegmentSelector(SRC); FI;
    DEST ←Offset(SRC);
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #UD | If source operand is not a memory location. |
| | If the LOCK prefix is used. |
| #GP(0) | If a NULL selector is loaded into the SS register. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #GP(selector) | If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL. |
| | If the DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #SS(selector) | If the SS register is being loaded and the segment is marked not present. |
| #NP(selector) | If DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If source operand is not a memory location. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | If source operand is not a memory location. |
| | If the LOCK prefix is used. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the memory address is in a non-canonical form. |
| | If a NULL selector is attempted to be loaded into the SS register in compatibility mode. |
| | If a NULL selector is attempted to be loaded into the SS register in CPL3 and 64-bit mode. |
| | If a NULL selector is attempted to be loaded into the SS register in non-CPL3 and 64-bit mode where its RPL is not equal to CPL. |
| #GP(Selector) | If the FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the memory address of the descriptor is non-canonical, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL. |
| | If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the memory address of the descriptor is non-canonical, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL. |
| #SS(0) | If a memory operand effective address is non-canonical |
| #SS(Selector) | If the SS register is being loaded and the segment is marked not present. |

| | |
|---|---|
| #NP(selector) | If FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If source operand is not a memory location. |
| | If the LOCK prefix is used. |

# LEA—Load Effective Address

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 8D /r | LEA *r16,m* | Valid | Valid | Store effective address for *m* in register *r16.* |
| 8D /r | LEA *r32,m* | Valid | Valid | Store effective address for *m* in register *r32.* |
| REX.W + 8D /r | LEA *r64,m* | Valid | N.E. | Store effective address for *m* in register *r64.* |

## Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

**Table 3-58. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes**

| Operand Size | Address Size | Action Performed |
|---|---|---|
| 16 | 16 | 16-bit effective address is calculated and stored in requested 16-bit register destination. |
| 16 | 32 | 32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination. |
| 32 | 16 | 16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination. |
| 32 | 32 | 32-bit effective address is calculated and stored in the requested 32-bit register destination. |

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

In 64-bit mode, the instruction's destination operand is governed by operand size attribute, the default operand size is 32 bits. Address calculation is governed by address size attribute, the default address size is 64-bits. In 64-bit mode, address size of 16 bits is not encodable. See Table 3-59.

### Table 3-59. 64-bit Mode LEA Operation with Address and Operand Size Attributes

| Operand Size | Address Size | Action Performed |
|---|---|---|
| 16 | 32 | 32-bit effective address is calculated (using 67H prefix). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix). |
| 16 | 64 | 64-bit effective address is calculated (default address size). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix). |
| 32 | 32 | 32-bit effective address is calculated (using 67H prefix) and stored in the requested 32-bit register destination. |
| 32 | 64 | 64-bit effective address is calculated (default address size) and the lower 32 bits of the address are stored in the requested 32-bit register destination. |
| 64 | 32 | 32-bit effective address is calculated (using 67H prefix), zero-extended to 64-bits, and stored in the requested 64-bit register destination (using REX.W). |
| 64 | 64 | 64-bit effective address is calculated (default address size) and all 64-bits of the address are stored in the requested 64-bit register destination (using REX.W). |

## Operation

```
IF OperandSize = 16 and AddressSize = 16
    THEN
        DEST ←EffectiveAddress(SRC); (* 16-bit address *)
    ELSE IF OperandSize = 16 and AddressSize = 32
        THEN
            temp ←EffectiveAddress(SRC); (* 32-bit address *)
            DEST ←temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 16
        THEN
            temp ←EffectiveAddress(SRC); (* 16-bit address *)
            DEST ←ZeroExtend(temp); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 32
        THEN
            DEST ←EffectiveAddress(SRC); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 16 and AddressSize = 64
        THEN
            temp ←EffectiveAddress(SRC); (* 64-bit address *)
```

```
            DEST ←temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 64
        THEN
            temp ←EffectiveAddress(SRC); (* 64-bit address *)
            DEST ←temp[0:31]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 64 and AddressSize = 64
        THEN
            DEST ←EffectiveAddress(SRC); (* 64-bit address *)
        FI;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #UD | If source operand is not a memory location. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# LEAVE—High Level Procedure Exit

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| C9 | LEAVE | Valid | Valid | Set SP to BP, then pop BP. |
| C9 | LEAVE | N.E. | Valid | Set ESP to EBP, then pop EBP. |
| C9 | LEAVE | Valid | N.E. | Set RSP to RBP, then pop RBP. |

## Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See "Procedure Calls for Block-Structured Languages" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on the use of the ENTER and LEAVE instructions.

In 64-bit mode, the instruction's default operation size is 64 bits; 32-bit operation cannot be encoded. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF StackAddressSize = 32
    THEN
        ESP ←EBP;
    ELSE IF StackAddressSize = 64
        THEN RSP ←RBP; FI;
    ELSE IF StackAddressSize = 16
        THEN SP ←BP; FI;
FI;

IF OperandSize = 32
    THEN EBP ←Pop();
    ELSE IF OperandSize = 64
        THEN RBP ←Pop(); FI;
    ELSE IF OperandSize = 16
        THEN BP ←Pop(); FI;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #SS(0) | If the EBP register points to a location that is not within the limits of the current stack segment. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If the EBP register points to a location outside of the effective address space from 0 to FFFFH. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If the EBP register points to a location outside of the effective address space from 0 to FFFFH. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If the stack address is in a non-canonical form. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# LFENCE—Load Fence

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F AE /5 | LFENCE | Valid | Valid | Serializes load operations. |

## Description

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. This serializing operation guarantees that every load instruction that precedes in program order the LFENCE instruction is globally visible before any load instruction that follows the LFENCE instruction is globally visible. The LFENCE instruction is ordered with respect to load instructions, other LFENCE instructions, any MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to store instructions or the SFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue and speculative reads. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The LFENCE instruction provides a performance-efficient way of insuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). The PREFETCH*h* instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the LFENCE instruction is not ordered with respect to PREFETCH*h* instructions or any other speculative fetching mechanism (that is, data could be speculative loaded into the cache just before, during, or after the execution of an LFENCE instruction).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
Wait_On_Following_Loads_Until(preceding_loads_globally_visible);
```

## Intel C/C++ Compiler Intrinsic Equivalent

void _mm_lfence(void)

## Exceptions (All Modes of Operation)

| | |
|---|---|
| #UD | If the LOCK prefix is used. |

# LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 01 /2 | LGDT *m16&32* | N.E. | Valid | Load *m* into GDTR. |
| 0F 01 /3 | LIDT *m16&32* | N.E. | Valid | Load *m* into IDTR. |
| 0F 01 /2 | LGDT *m16&64* | Valid | N.E. | Load *m* into GDTR. |
| 0F 01 /3 | LIDT *m16&64* | Valid | N.E. | Load *m* into IDTR. |

## Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

In 64-bit mode, the instruction's operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

See "SGDT—Store Global Descriptor Table Register" in Chapter 4, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for information on storing the contents of the GDTR and IDTR.

## Operation

```
IF Instruction is LIDT
    THEN
        IF OperandSize = 16
            THEN
                IDTR(Limit) ←SRC[0:15];
                IDTR(Base) ←SRC[16:47] AND 00FFFFFFH;
            ELSE IF 32-bit Operand Size
                THEN
```

```
                    IDTR(Limit) ←SRC[0:15];
                    IDTR(Base) ←SRC[16:47];
                FI;
            ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
                THEN
                    IDTR(Limit) ←SRC[0:15];
                    IDTR(Base) ←SRC[16:79];
                FI;
        FI;
    ELSE (* Instruction is LGDT *)
        IF OperandSize = 16
            THEN
                GDTR(Limit) ←SRC[0:15];
                GDTR(Base) ←SRC[16:47] AND 00FFFFFFH;
            ELSE IF 32-bit Operand Size
                THEN
                    GDTR(Limit) ←SRC[0:15];
                    GDTR(Base) ←SRC[16:47];
                FI;
            ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
                THEN
                    GDTR(Limit) ←SRC[0:15];
                    GDTR(Base) ←SRC[16:79];
                FI;
        FI;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #UD | If source operand is not a memory location. |
| | If the LOCK prefix is used. |
| #GP(0) | If the current privilege level is not 0. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If source operand is not a memory location. |
| | If the LOCK prefix is used. |
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | If source operand is not a memory location. |
| | If the LOCK prefix is used. |
| #GP(0) | The LGDT and LIDT instructions are not recognized in virtual-8086 mode. |
| #GP | If the current privilege level is not 0. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the current privilege level is not 0. |
| | If the memory address is in a non-canonical form. |
| #UD | If source operand is not a memory location. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | If a page fault occurs. |

# LLDT—Load Local Descriptor Table Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 00 /2 | LLDT *r/m*16 | Valid | Valid | Load segment selector *r/m*16 into LDTR. |

## Description

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses the segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If bits 2-15 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes silently. However, all subsequent references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. This instruction can only be executed in protected mode or 64-bit mode.

In 64-bit mode, the operand size is fixed at 16 bits.

## Operation

```
IF SRC(Offset) > descriptor table limit
    THEN #GP(segment selector); FI;

IF segment selector is valid

    Read segment descriptor;

    IF SegmentDescriptor(Type) ≠ LDT
        THEN #GP(segment selector); FI;
    IF segment descriptor is not present
        THEN #NP(segment selector); FI;

    LDTR(SegmentSelector) ←SRC;
    LDTR(SegmentDescriptor) ←GDTSegmentDescriptor;
```

```
ELSE LDTR ←INVALID
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #GP(selector) | If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table. |
| | Segment selector is beyond GDT limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NP(selector) | If the LDT descriptor is not present. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | The LLDT instruction is not recognized in real-address mode. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | The LLDT instruction is not recognized in virtual-8086 mode. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the current privilege level is not 0. |
| | If the memory address is in a non-canonical form. |
| #GP(selector) | If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table. |
| | Segment selector is beyond GDT limit. |
| #NP(selector) | If the LDT descriptor is not present. |

| | |
|---|---|
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

# LMSW—Load Machine Status Word

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 01 /6 | LMSW *r/m*16 | Valid | Valid | Loads *r/m*16 in machine status word of CR0. |

## Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used to clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Note that the operand size is fixed at 16 bits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

```
CR0[0:3] ←SRC[0:3];
```

## Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the current privilege level is not 0. |
| | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

# LOCK—Assert LOCK# Signal Prefix

| Opcode* | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F0 | LOCK | Valid | Valid | Asserts LOCK# signal for duration of the accompanying instruction. |

**NOTES:**

* See IA-32 Architecture Compatibility section below.

## Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal insures that the processor has exclusive use of any shared memory while the signal is asserted.

Note that, in later Intel 64 and IA-32 processors (including the Pentium 4, Intel Xeon, and P6 family processors), locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below.

The LOCK prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. If the LOCK prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception (#UD) may be generated. An undefined opcode exception will also be generated if the LOCK prefix is used with any instruction not in the above list. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## IA-32 Architecture Compatibility

Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism insures that the operation is carried out atomically with regards to memory. See "Effects of a Locked Operation on Internal Processor Caches" in Chapter 7 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the for more information on locking of caches.

## Operation

```
AssertLOCK#(DurationOfAccompaningInstruction);
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #UD | If the LOCK prefix is used with an instruction not listed: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG.<br>Other exceptions can be generated by the instruction when the LOCK prefix is applied. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# LODS/LODSB/LODSW/LODSD/LODSQ—Load String

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| AC | LODS *m8* | Valid | Valid | For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL. |
| AD | LODS *m16* | Valid | Valid | For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX. |
| AD | LODS m32 | Valid | Valid | For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX. |
| REX.W + AD | LODS *m64* | Valid | N.E. | Load qword at address (R)SI into RAX. |
| AC | LODSB | Valid | Valid | For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL. |
| AD | LODSW | Valid | Valid | For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX. |
| AD | LODSD | Valid | Valid | For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX. |
| REX.W + AD | LODSQ | Valid | N.E. | Load qword at address (R)SI into RAX. |

## Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:EDI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The DS segment may be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the

correct **location**. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. LODS/LODSQ load the quadword at address (R)SI into RAX. The (R)SI register is then incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See "REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix.

## Operation

```
IF AL ←SRC; (* Byte load *)
    THEN AL ←SRC; (* Byte load *)
        IF DF = 0
            THEN (E)SI ←(E)SI +1;
            ELSE (E)SI ←(E)SI – 1;
        FI;
ELSE IF AX ←SRC; (* Word load *)
    THEN IF DF = 0
            THEN (E)SI ←(E)SI +2;
            ELSE (E)SI ←(E)SI – 2;
        IF;
    FI;
ELSE IF EAX ←SRC; (* Doubleword load *)
    THEN IF DF = 0
            THEN (E)SI ←(E)SI +4;
            ELSE (E)SI ←(E)SI – 4;
```

```
        FI;
    FI;
ELSE IF RAX ←SRC; (* Quadword load *)
    THEN IF DF = 0
            THEN (R)SI ←(R)SI +8;
            ELSE (R)SI ←(R)SI – 8;
        FI;
    FI;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# LOOP/LOOP*cc*—Loop According to ECX Counter

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| E2 *cb* | LOOP *rel8* | Valid | Valid | Decrement count; jump short if count ≠ 0. |
| E1 *cb* | LOOPE *rel8* | Valid | Valid | Decrement count; jump short if count ≠ 0 and ZF = 1. |
| E0 *cb* | LOOPNE *rel8* | Valid | Valid | Decrement count; jump short if count ≠ 0 and ZF = 0. |

## Description

Performs a loop operation using the RCX, ECX or CX register as a counter (depending on whether address size is 64 bits, 32 bits, or 16 bits). Note that the LOOP instruction ignores REX.W; but 64-bit address size can be over-ridden using a 67H prefix.

Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the IP/EIP/RIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP*cc* instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

## Operation

```
IF (AddressSize = 32)
    THEN Count is ECX;
ELSE IF (AddressSize = 64)
    Count is RCX;
ELSE Count is CX;
FI;

Count ←Count – 1;

IF Instruction is not LOOP
    THEN
```

```
        IF (Instruction ←LOOPE) or (Instruction ←LOOPZ)
            THEN IF (ZF = 1) and (Count ≠ 0)
                    THEN BranchCond ←1;
                    ELSE BranchCond ←0;
                FI;
            ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
                IF (ZF = 0 ) and (Count ≠ 0)
                    THEN BranchCond ←1;
                    ELSE BranchCond ←0;
                FI;
        FI;
    ELSE (* Instruction = LOOP *)
        IF (Count ≠ 0)
            THEN BranchCond ←1;
            ELSE BranchCond ←0;
        FI;
FI;

IF BranchCond = 1
    THEN
        IF OperandSize = 32
            THEN EIP ←EIP +SignExtend(DEST);
            ELSE IF OperandSize = 64
                THEN RIP ←RIP +SignExtend(DEST);
                FI;
            ELSE IF OperandSize = 16
                THEN EIP ←EIP AND 0000FFFFH;
                FI;
            ELSE IF OperandSize = (32 or 64)
                THEN IF (R/E)IP < CS.Base or (R/E)IP >CS.Limit
                    #GP; FI;
                FI;
        FI;
    ELSE
        Terminate loop and continue program execution at (R/E)IP;
FI;
```

## Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the offset being jumped to is beyond the limits of the CS segment. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the offset being jumped to is in a non-canonical form. |
| #UD | If the LOCK prefix is used. |

# LSL—Load Segment Limit

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 03 /r | LSL *r16, r16/m16* | Valid | Valid | Load: *r16* ←segment limit, selector *r16/m16.* |
| 0F 03 /r | LSL *r32, r32/m16** | Valid | Valid | Load: *r32* ←segment limit, selector *r32/m16.* |
| REX.W + 0F 03 /r | LSL *r64, r32/m16** | Valid | Valid | Load: *r64* ←segment limit, selector *r32/m16* |

**NOTES:**

* For all loads (regardless of destination sizing), only bits 16-0 are used. Other bits are ignored.

## Description

Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit “raw” limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed

- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

**Table 3-60. Segment and Gate Descriptor Types**

| Type | Protected Mode | | IA-32e Mode | |
|---|---|---|---|---|
| | Name | Valid | Name | Valid |
| 0 | Reserved | No | Upper 8 byte of a 16-Byte descriptor | Yes |
| 1 | Available 16-bit TSS | Yes | Reserved | No |
| 2 | LDT | Yes | LDT | Yes |
| 3 | Busy 16-bit TSS | Yes | Reserved | No |
| 4 | 16-bit call gate | No | Reserved | No |
| 5 | 16-bit/32-bit task gate | No | Reserved | No |
| 6 | 16-bit interrupt gate | No | Reserved | No |
| 7 | 16-bit trap gate | No | Reserved | No |
| 8 | Reserved | No | Reserved | No |
| 9 | Available 32-bit TSS | Yes | 64-bit TSS | Yes |
| A | Reserved | No | Reserved | No |
| B | Busy 32-bit TSS | Yes | Busy 64-bit TSS | Yes |
| C | 32-bit call gate | No | 64-bit call gate | No |
| D | Reserved | No | Reserved | No |
| E | 32-bit interrupt gate | No | 64-bit interrupt gate | No |
| F | 32-bit trap gate | No | 64-bit trap gate | No |

## Operation

```
IF SRC(Offset) > descriptor table limit
    THEN ZF ←0; FI;
```

```
Read segment descriptor;

IF SegmentDescriptor(Type) ≠ conforming code segment
and (CPL > DPL) OR (RPL > DPL)
or Segment type is not valid for instruction
        THEN
            ZF ←0;
        ELSE
            temp ←SegmentLimit([SRC]);
            IF (G ←1)
                THEN temp ←ShiftLeft(12, temp) OR 00000FFFH;
            ELSE IF OperandSize = 32
                THEN DEST ←temp; FI;
            ELSE IF OperandSize = 64 (* REX.W used *)
                THEN DEST (* Zero-extended *) ←temp; FI;
            ELSE (* OperandSize = 16 *)
                DEST ←temp AND FFFFH;
            FI;
FI;
```

## Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is set to 0.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | The LAR instruction cannot be executed in real-address mode. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | The LAR instruction cannot be executed in virtual-8086 mode. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If the memory operand effective address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory operand effective address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# LTR—Load Task Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 00 /3 | LTR *r/m*16 | Valid | Valid | Load *r/m*16 into task register. |

## Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

## Operation

```
IF SRC is a NULL selector
    THEN #GP(0);

IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
    THEN #GP(segment selector); FI;

Read segment descriptor;

IF segment descriptor is not for an available TSS
    THEN #GP(segment selector); FI;
IF segment descriptor is not present
    THEN #NP(segment selector); FI;

TSSsegmentDescriptor(busy) ←1;
(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)

TaskRegister(SegmentSelector) ←SRC;
TaskRegister(SegmentDescriptor) ←TSSSegmentDescriptor;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the source operand contains a NULL segment selector. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| #GP(selector) | If the source selector points to a segment that is not a TSS or to one for a task that is already busy. |
| | If the selector points to LDT or is beyond the GDT limit. |
| #NP(selector) | If the TSS is marked not present. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | The LTR instruction is not recognized in real-address mode. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #UD | The LTR instruction is not recognized in virtual-8086 mode. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the current privilege level is not 0. |
| | If the memory address is in a non-canonical form. |
| | If the source operand contains a NULL segment selector. |
| #GP(selector) | If the source selector points to a segment that is not a TSS or to one for a task that is already busy. |
| | If the selector points to LDT or is beyond the GDT limit. |
| | If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero. |

| | |
|---|---|
| #NP(selector) | If the TSS is marked not present. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

# MASKMOVDQU—Store Selected Bytes of Double Quadword

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F F7 /r | MASKMOVDQU *xmm1, xmm2* | Valid | Valid | Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:EDI. |

## Description

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The location of the first byte of the memory location is specified by DI/EDI and DS registers. The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVEDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVEDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

Behavior with a mask of all 0s is as follows:

- No data will be written to memory.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVDQU instruction can be used to improve performance of algorithms that need to merge data on a byte-by-byte basis. MASKMOVDQU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
IF (MASK[7] = 1)
    THEN DEST[DI/EDI]  ←SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ←SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 14th bytes in source operand *)
IF (MASK[127] = 1)
    THEN DEST[DI/EDI +15] ←SRC[127:120] ELSE (* Memory location unchanged *); FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. (even if mask is all 0s). |
| | If the destination operand is in a nonwritable segment. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | For an illegal address in the SS segment (even if mask is all 0s). |
| #PF(fault-code) | For a page fault (implementation specific). |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. (even if mask is all 0s). |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault (implementation specific). |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the memory address is in a non-canonical form. |
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #PF(fault-code) | For a page fault (implementation specific). |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# MASKMOVQ—Store Selected Bytes of Quadword

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F F7 /r | MASKMOVQ *mm1, mm2* | Valid | Valid | Selectively write bytes from *mm1* to memory location using the byte mask in *mm2.* The default memory location is specified by DS:EDI. |

## Description

Stores selected bytes from the source operand (first operand) into a 64-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are MMX technology registers. The location of the first byte of the memory location is specified by DI/EDI and DS registers. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQ instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVEDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction causes a transition from x87 FPU to MMX technology state (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]).

The behavior of the MASKMOVQ instruction with a mask of all 0s is as follows:

- No data will be written to memory.
- Transition from x87 FPU to MMX technology state will occur.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- Signaling of breakpoints (code or data) is not guaranteed (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, the memory address is specified by DS:RDI.

## Operation

```
IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ←SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ←SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 6th bytes in source operand *)
IF (MASK[63] = 1)
    THEN DEST[DI/EDI +15] ←SRC[63:56] ELSE (* Memory location unchanged *); FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmove_si64(__m64d, __m64n, char * p)

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments (even if mask is all 0s).<br>If the destination operand is in a nonwritable segment.<br>If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | For an illegal address in the SS segment (even if mask is all 0s). |
| #PF(fault-code) | For a page fault (implementation specific). |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending FPU exception. |
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If Mod field of the ModR/M byte not 11B.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. (even if mask is all 0s). |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #MF | If there is a pending FPU exception. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault (implementation specific). |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the memory address is in a non-canonical form. |
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #PF(fault-code) | For a page fault (implementation specific). |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending FPU exception. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If Mod field of the ModR/M byte not 11B. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MAXPD—Return Maximum Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 5F /r | MAXPD *xmm1, xmm2/m128* | Valid | Valid | Return the maximum double-precision floating-point values between *xmm2/m128* and *xmm1*. |

## Description

Performs a SIMD compare of the packed double-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of values to the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←        IF ((DEST[63:0] = 0.0) and (SRC[63:0] = 0.0))
                        THEN SRC[63:0];
                        ELSE IF (DEST[63:0] = SNaN) THEN SRC[63:0]; FI;
                        ELSE IF (SRC[63:0] = SNaN) THEN SRC[63:0]; FI;
                        ELSE IF (DEST[63:0] > SRC[63:0])
                            THEN DEST[63:0];
                            ELSE SRC[63:0]; FI; FI;

DEST[127:64] ←      IF ((DEST[127:64] = 0.0) and (SRC[127:64] = 0.0))
                        THEN SRC[127:64];
                        ELSE IF (DEST[127:64] = SNaN) THEN SRC[127:64]; FI;
                        ELSE IF (SRC[127:64] = SNaN) THEN SRC[127:64]; FI;
                        ELSE IF (DEST[127:64] > SRC[63:0])
                            THEN DEST[127:64];
```

```
ELSE SRC[127:64]; FI; FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MAXPD __m128d _mm_max_pd(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# MAXPS—Return Maximum Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 5F /r | MAXPS *xmm1, xmm2/m128* | Valid | Valid | Return the maximum single-precision floating-point values between *xmm2/m128* and *xmm1*. |

## Description

Performs a SIMD compare of the packed single-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of values to the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←        IF ((DEST[31:0] = 0.0) and (SRC[31:0] = 0.0))
                        THEN SRC[31:0];
                        ELSE IF (DEST[31:0] = SNaN) THEN SRC[31:0]; FI;
                        ELSE IF (SRC[31:0] = SNaN) THEN SRC[31:0]; FI;
                        ELSE IF (DEST[31:0] > SRC[31:0]); FI;
                            THEN DEST[31:0];
                            ELSE SRC[31:0]; FI; FI;
(* Repeat operation for 2nd and 3rd doublewords *);

DEST[127:64] ←      IF ((DEST[127:96] = 0.0) and (SRC[127:96] = 0.0))
                        THEN SRC[127:96];
                        ELSE IF (DEST[127:96] = SNaN) THEN SRC[127:96];
                        ELSE IF (SRC[127:96] = SNaN) THEN SRC[127:96];
                        ELSE IF (DEST[127:96] > SRC[127:96])
```

```
        THEN DEST[127:96];
        ELSE SRC[127:96]; FI; FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MAXPS __m128d _mm_max_ps(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |

| | |
|---|---|
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXM-MEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXM-MEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

# MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 5F /r | MAXSD *xmm1, xmm2/m64* | Valid | Valid | Return the maximum scalar double-precision floating-point value between *xmm2/mem64* and *xmm1*. |

## Description

Compares the low double-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the maximum value to the low quadword of the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is a memory operand, only 64 bits are accessed. The high quadword of the destination operand remains unchanged.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←        IF ((DEST[63:0] = 0.0) and (SRC[63:0] = 0.0))
                        THEN SRC[63:0]; FI;
                    IF (DEST[63:0] = SNaN)
                        THEN SRC[63:0];
                        ELSE IF (SRC[63:0] = SNaN)
                            THEN SRC[63:0]; FI;
                        ELSE IF (DEST[63:0] > SRC[63:0])
                            THEN DEST[63:0];
                            ELSE SRC[63:0]; FI; FI;

(* DEST[127:64] is unchanged *);
```

## Intel C/C++ Compiler Intrinsic Equivalent

MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 5F /r | MAXSS *xmm1, xmm2/m32* | Valid | Valid | Return the maximum scalar single-precision floating-point value between *xmm2/mem32* and *xmm1*. |

## Description

Compares the low single-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the maximum value to the low doubleword of the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. When the source operand is a memory operand, only 32 bits are accessed. The three high-order doublewords of the destination operand remain unchanged.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ← IF ((DEST[31:0] = 0.0) and (SRC[31:0] = 0.0))
                THEN SRC[31:0];
                ELSE IF (DEST[31:0] = SNaN) THEN SRC[31:0]; FI;
                ELSE IF (SRC[31:0] = SNaN) THEN SRC[31:0]; FI;
                ELSE IF (DEST[31:0] > SRC[31:0])
                    THEN DEST[31:0]
                    ELSE SRC[31:0]; FI; FI;
(* DEST[127:32] is unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

__m128d _mm_max_ss(__m128d a, __m128d b)

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MFENCE—Memory Fence

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F AE /6 | MFENCE | Valid | Valid | Serializes load and store operations. |

## Description

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes in program order the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible. The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any SFENCE and LFENCE instructions, and any serializing instructions (such as the CPUID instruction).

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). The PREFETCH*h* instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the MFENCE instruction is not ordered with respect to PREFETCH*h* instructions or any other speculative fetching mechanism (that is, data could be speculatively loaded into the cache just before, during, or after the execution of an MFENCE instruction).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
Wait_On_Following_Loads_And_Stores_Until(preceding_loads_and_stores_globally_visible);
```

## Intel C/C++ Compiler Intrinsic Equivalent

void _mm_mfence(void)

## Exceptions (All Modes of Operation)

| | |
|---|---|
| #UD | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# MINPD—Return Minimum Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 5D /r | MINPD *xmm1, xmm2/m128* | Valid | Valid | Return the minimum double-precision floating-point values between *xmm2/m128* and *xmm1*. |

## Description

Performs a SIMD compare of the packed double-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of values to the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←        IF ((DEST[63:0] = 0.0) and (SRC[63:0] = 0.0))
                        THEN SRC[63:0]; FI;
                        ELSE IF (DEST[63:0] = SNaN) THEN SRC[63:0]; FI;
                        ELSE IF (SRC[63:0] = SNaN) THEN SRC[63:0]; FI;
                        ELSE IF (DEST[63:0] < SRC[63:0])
                            THEN DEST[63:0]
                            ELSE SRC[63:0]; FI; FI;

DEST[127:64] ←      IF ((DEST[127:64] = 0.0) and (SRC[127:64] = 0.0))
                        THEN SRC[127:64]; FI;
                        ELSE IF (DEST[127:64] = SNaN) THEN SRC[127:64]; FI;
                        ELSE IF (SRC[127:64] = SNaN) THEN SRC[127:64]; FI;
                        ELSE IF (DEST[127:64] < SRC[63:0])
```

```
        THEN DEST[127:64]
        ELSE SRC[127:64]; FI; FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MINPD __m128d _mm_min_pd(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |

| | |
|---|---|
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXM-MEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXM-MEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# MINPS—Return Minimum Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 5D /r | MINPS *xmm1, xmm2/m128* | Valid | Valid | Return the minimum single-precision floating-point values between *xmm2/m128* and *xmm1*. |

## Description

Performs a SIMD compare of the packed single-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of values to the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←        IF ((DEST[31:0] = 0.0) and (SRC[31:0] = 0.0))
                        THEN SRC[31:0];
                        ELSE IF (DEST[31:0] = SNaN) THEN SRC[31:0]; FI;
                        ELSE IF (SRC[31:0] = SNaN) THEN SRC[31:0]; FI;
                        ELSE IF (DEST[31:0] >SRC[31:0])
                            THEN DEST[31:0]
                            ELSE SRC[31:0]; FI; FI;
(* Repeat operation for 2nd and 3rd doublewords *);

DEST[127:64] ←      IF ((DEST127:96] = 0.0) and (SRC[127:96] = 0.0))
                        THEN SRC[127:96];
                        ELSE IF (DEST[127:96] = SNaN) THEN SRC[127:96]; FI;
                        ELSE IF (SRC[127:96] = SNaN) THEN SRC[127:96]; FI;
                        ELSE IF (DEST[127:96] < SRC[127:96])
```

```
        THEN DEST[127:96]
        ELSE SRC[127:96]; FI; FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MINPS __m128d _mm_min_ps(__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |

| | |
|---|---|
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

# MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 5D /r | MINSD *xmm1, xmm2/m64* | Valid | Valid | Return the minimum scalar double-precision floating-point value between *xmm2/mem64* and *xmm1*. |

## Description

Compares the low double-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the minimum value to the low quadword of the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. When the source operand is a memory operand, only the 64 bits are accessed. The high quadword of the destination operand remains unchanged.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←        IF ((DEST[63:0] = 0.0) and (SRC[63:0] = 0.0))
                        THEN SRC[63:0];
                        ELSE IF (DEST[63:0] = SNaN) THEN SRC[63:0]; FI;
                        ELSE IF (SRC[63:0] = SNaN) THEN SRC[63:0]; FI;
                        ELSE IF (DEST[63:0] < SRC[63:0])
                            THEN DEST[63:0];
                            ELSE SRC[63:0]; FI; FI;

(* DEST[127:64] is unchanged *);
```

## Intel C/C++ Compiler Intrinsic Equivalent

MINSD __m128d _mm_min_sd(__m128d a, __m128d b)

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 5D /r | MINSS *xmm1, xmm2/m32* | Valid | Valid | Return the minimum scalar single-precision floating-point value between *xmm2/mem32* and *xmm1*. |

## Description

Compares the low single-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the minimum value to the low doubleword of the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. When the source operand is a memory operand, only 32 bits are accessed. The three high-order doublewords of the destination operand remain unchanged.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←        IF ((DEST[31:0] = 0.0) AND (SRC[31:0] = 0.0))
                        THEN SRC[31:0];
                        ELSE IF (DEST[31:0] = SNaN) THEN SRC[31:0]; FI;
                        ELSE IF (SRC[31:0] = SNaN) THEN SRC[31:0]; FI;
                        ELSE IF (DEST[31:0] < SRC[31:0])
                            THEN DEST[31:0]
                            ELSE SRC[31:0]; FI; FI;

(* DEST[127:32] is unchanged *);
```

## Intel C/C++Compiler Intrinsic Equivalent

MINSS __m128d _mm_min_ss(__m128d a, __m128d b)

### SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MONITOR—Set Up Monitor Address

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| OF 01 *C8* | MONITOR | Valid | Valid | Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The default address is DS:EAX. |

## Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The content of EAX is an effective address. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 7, “Multiple-Processor Management” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction can be used at all privilege levels and is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

The MONITOR CPUID feature flag (ECX bit 3; CPUID executed EAX = 1) indicates the availability of MONITOR and MWAIT in the processor. When set, the unconditional execution of MONITOR is supported at privilege levels 0; conditional execution is supported at privilege levels 1 through 3 (test for the appropriate support before unconditional use). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLES MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an illegal opcode exception.

The instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

## Intel C/C++ Compiler Intrinsic Equivalent

MONITOR void _mm_monitor(void const *p, unsigned extensions,unsigned hints)

## Numeric Exceptions

None

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| | If ECX ←0. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |
| | If current privilege level is not 0. |

## Real Address Mode Exceptions

| | |
|---|---|
| #GP | If any part of the operand in the CS, DS, ES, FS, or GS segment lies outside of the effective address space from 0 to FFFFH. |
| | If ECX ←0. |
| #SS | If any part of the operand in the SS segment lies outside of the effective address space from 0 to FFFFH. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |

## Virtual 8086 Mode Exceptions

| | |
|---|---|
| #UD | The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1). |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form.<br>If RCX ←0. |
| #SS(0) | If the linear address of the operand in the SS segment is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #UD | If the current privilege level is not 0.<br>If CPUID.01H:ECX.MONITOR[bit 3] = 0. |

## MOV—Move

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 88 /r | MOV *r/m8,r8* | Valid | Valid | Move *r8* to *r/m8.* |
| REX + 88 /r | MOV *r/m8****,*r8**** | Valid | N.E. | Move *r8* to *r/m8.* |
| 89 /r | MOV *r/m16,r16* | Valid | Valid | Move *r16* to *r/m16.* |
| 89 /r | MOV *r/m32,r32* | Valid | Valid | Move *r32* to *r/m32.* |
| REX.W + 89 /r | MOV *r/m64,r64* | Valid | N.E. | Move *r64* to *r/m64.* |
| 8A /r | MOV *r8,r/m8* | Valid | Valid | Move *r/m8* to *r8.* |
| REX + 8A /r | MOV *r8****,*r/m8**** | Valid | N.E. | Move *r/m8* to *r8.* |
| 8B /r | MOV *r16,r/m16* | Valid | Valid | Move *r/m16* to *r16.* |
| 8B /r | MOV *r32,r/m32* | Valid | Valid | Move *r/m32* to *r32.* |
| REX.W + 8B /r | MOV *r64,r/m64* | Valid | N.E. | Move *r/m64* to *r64.* |
| 8C /r | MOV *r/m16,Sreg*** | Valid | Valid | Move segment register to *r/m16.* |
| REX.W + 8C /r | MOV *r/m64,Sreg*** | Valid | Valid | Move zero extended 16-bit segment register to *r/m64.* |
| 8E /r | MOV *Sreg,r/m16*** | Valid | Valid | Move *r/m16* to segment register. |
| REX.W + 8E /r | MOV *Sreg,r/m64*** | Valid | Valid | Move *lower 16 bits of r/m64* to segment register. |
| A0 | MOV AL,*moffs8** | Valid | Valid | Move byte at (*seg:offset*) to AL. |
| REX.W + A0 | MOV AL,*moffs8** | Valid | N.E. | Move byte at (*offset*) to AL. |
| A1 | MOV AX,*moffs16** | Valid | Valid | Move word at (*seg:offset*) to AX. |
| A1 | MOV EAX,*moffs32** | Valid | Valid | Move doubleword at (*seg:offset*) to EAX. |
| REX.W + A1 | MOV RAX,*moffs64** | Valid | N.E. | Move quadword at (*offset*) to RAX. |
| A2 | MOV *moffs8*,AL | Valid | Valid | Move AL to (*seg:offset*). |
| REX.W + A2 | MOV *moffs8****,AL | Valid | N.E. | Move AL to (*offset*). |
| A3 | MOV *moffs16**,AX | Valid | Valid | Move AX to (*seg:offset*). |
| A3 | MOV *moffs32**,EAX | Valid | Valid | Move EAX to (*seg:offset*). |

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| REX.W + A3 | MOV *moffs64**,RAX | Valid | N.E. | Move RAX to (*offset*). |
| B0+ *rb* | MOV *r8, imm8* | Valid | Valid | Move *imm8* to *r8.* |
| REX + B0+ *rb* | MOV *r8****, *imm8* | Valid | N.E. | Move *imm8* to *r8.* |
| B8+ *rw* | MOV *r16, imm16* | Valid | Valid | Move *imm16* to *r16.* |
| B8+ *rd* | MOV *r32, imm32* | Valid | Valid | Move *imm32* to *r32.* |
| REX.W + B8+ *rd* | MOV *r64, imm64* | Valid | N.E. | Move *imm64* to *r64.* |
| C6 */0* | MOV *r/m8, imm8* | Valid | Valid | Move *imm8* to *r/m8.* |
| REX + C6 */0* | MOV *r/m8****, *imm8* | Valid | N.E. | Move *imm8* to *r/m8.* |
| C7 */0* | MOV *r/m16, imm16* | Valid | Valid | Move *imm16* to *r/m16.* |
| C7 */0* | MOV *r/m32, imm32* | Valid | Valid | Move *imm32* to *r/m32.* |
| REX.W + C7 */0* | MOV *r/m64, imm32* | Valid | N.E. | Move *imm32 sign extended to 64-bits* to *r/m64.* |

**NOTES:**

* The *moffs8*, *moffs16*, *moffs32* and *moffs64* operands specify a simple offset relative to the segment base, where 8, 16, 32 and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32 or 64 bits.

** In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following "Description" section for further information).

***In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, or a doubleword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the "Operation" algorithm

below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A NULL segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction inhibits all interrupts until after the execution of the next instruction. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an interrupt occurs[1]. Be aware that the LSS instruction offers a more efficient method of loading the SS and ESP registers.

When operating in 32-bit mode and moving data between a segment register and a general-purpose register, the 32-bit IA-32 processors do not require the use of the 16-bit operand-size prefix (a byte with the value 66H) with this instruction, but most assemblers will insert it if the standard form of the instruction is used (for example, MOV DS, AX). The processor will execute this instruction correctly, but it will usually require an extra clock. With most assemblers, using the instruction form MOV DS, EAX will avoid this unneeded 66H prefix. When the processor executes the instruction with a 32-bit general-purpose register, it assumes that the 16 least-significant bits of the general-purpose register are the destination or source operand. If the register is a destination operand, the resulting value in the two high-order bytes of the register is implementation dependent. For the Pentium 4, Intel Xeon, and P6 family processors, the two high-order bytes are filled with zeros; for earlier 32-bit IA-32 processors, the two high order bytes are undefined.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

DEST ←SRC;

---

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a MOV SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that load the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

   In the following sequence, interrupts may be recognized before MOV ESP, EBP executes:

   MOV SS, EDX
   MOV SS, EAX
   MOV ESP, EBP

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor to which it points.

```
IF SS is loaded
    THEN
        IF segment selector is NULL
            THEN #GP(0); FI;
        IF segment selector index is outside descriptor table limits
        or segment selector's RPL ≠ CPL
        or segment is not a writable data segment
        or DPL ≠ CPL
            THEN #GP(selector); FI;
        IF segment not marked present
            THEN #SS(selector);
            ELSE
                SS ←segment selector;
                SS ←segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector
THEN
    IF segment selector index is outside descriptor table limits
    or segment is not a data or readable code segment
    or ((segment is a data or nonconforming code segment)
    and (both RPL and CPL > DPL))
        THEN #GP(selector); FI;
    IF segment not marked present
        THEN #NP(selector);
        ELSE
            SegmentRegister ←segment selector;
            SegmentRegister ←segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with NULL selector
    THEN
        SegmentRegister ←segment selector;
        SegmentRegister ←segment descriptor;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If attempt is made to load SS register with NULL segment selector. |
| | If the destination operand is in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #GP(selector) | If segment selector index is outside descriptor table limits. |
| | If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL. |
| | If the SS register is being loaded and the segment pointed to is a non-writable data segment. |
| | If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment. |
| | If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #SS(selector) | If the SS register is being loaded and the segment pointed to is marked not present. |
| #NP | If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If attempt is made to load the CS register. |
| | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If attempt is made to load the CS register. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If attempt is made to load the CS register.<br>If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the memory address is in a non-canonical form.<br>If an attempt is made to load SS register with NULL segment selector when CPL = 3.<br>If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL. |
| #GP(selector) | If segment selector index is outside descriptor table limits.<br>If the memory access to the descriptor table is non-canonical.<br>If the SS register is being loaded and the segment selector's RPL and the segment descriptor’s DPL are not equal to the CPL.<br>If the SS register is being loaded and the segment pointed to is a nonwritable data segment.<br>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.<br>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL. |
| #SS(0) | If the stack address is in a non-canonical form. |
| #SS(selector) | If the SS register is being loaded and the segment pointed to is marked not present. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If attempt is made to load the CS register.<br>If the LOCK prefix is used. |

## MOV—Move to/from Control Registers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 20 /0 | MOV *r32*,CR0 | N.E. | Valid | Move CR0 to *r32*. |
| 0F 20 /0 | MOV *r64*,CR0 | Valid | N.E. | Move extended CR0 to *r64*. |
| 0F 20 /2 | MOV *r32*,CR2 | N.E. | Valid | Move CR2 to *r32*. |
| 0F 20 /2 | MOV *r64*,CR2 | Valid | N.E. | Move extended CR2 to *r64*. |
| 0F 20 /3 | MOV *r32*,CR3 | N.E. | Valid | Move CR3 to *r32*. |
| 0F 20 /3 | MOV *r64*,CR3 | Valid | N.E. | Move extended CR3 to *r64*. |
| 0F 20 /4 | MOV *r32*,CR4 | N.E. | Valid | Move CR4 to *r32*. |
| 0F 20 /4 | MOV *r64*,CR4 | Valid | N.E. | Move extended CR4 to *r64*. |
| REX.R + 0F 20 /0 | MOV *r64*,CR8 | Valid | N.E. | Move extended CR8 to *r64*.[1] |
| 0F 22 /0 | MOV CR0,*r32* | N.E. | Valid | Move *r32* to CR0. |
| 0F 22 /0 | MOV CR0,*r64* | Valid | N.E. | Move *r64* to extended CR0. |
| 0F 22 /2 | MOV CR2,*r32* | N.E. | Valid | Move *r32* to CR2. |
| 0F 22 /2 | MOV CR2,*r64* | Valid | N.E. | Move *r64* to extended CR2. |
| 0F 22 /3 | MOV CR3,*r32* | N.E. | Valid | Move *r32* to CR3. |
| 0F 22 /3 | MOV CR3,*r64* | Valid | N.E. | Move *r64* to extended CR3. |
| 0F 22 /4 | MOV CR4,*r32* | N.E. | Valid | Move *r32* to CR4. |
| 0F 22 /4 | MOV CR4,*r64* | Valid | N.E. | Move *r64* to extended CR4. |
| REX.R + 0F 22 /0 | MOV CR8,*r64* | Valid | N.E. | Move *r64* to extended CR8. |

**NOTE:**

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 7 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Description

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See "Control Registers" in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4's reserved bits will cause a general protection fault. Reserved bits in CR0

and CR3 remain clear after any load of those registers; attempts to set them have no impact. On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are always 11B. The *r/m* field specifies the general-purpose register loaded or read.

These instructions have the following side effect:

- When writing to control register CR3, all non-global TLB entries are flushed (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

The following side effects are implementation specific for the Pentium 4, Intel Xeon, and P6 processor family. Software should not depend on this functionality in all Intel 64 or IA-32 processors:

- When modifying any of the paging flags in the control registers (PE and PG in register CR0 and PGE, PSE, and PAE in register CR4), all TLB entries are flushed, including global entries.
- If the PG flag is set to 1 and control register CR4 is written to set the PAE flag to 1 (to enable the physical address extension mode), the pointers in the page-directory pointers table (PDPT) are loaded into the processor (into internal, non-architectural registers).
- If the PAE flag is set to 1 and the PG flag set to 1, writing to control register CR3 will cause the PDPTRs to be reloaded into the processor. If the PAE flag is set to 1 and control register CR0 is written to set the PG flag, the PDPTRs are reloaded into the processor.

In 64-bit mode, the instruction's default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8-R15). Use of the REX.W prefix or 66H prefix is ignored. See the summary chart at the beginning of this section for encoding data and limits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

```
DEST ←SRC;
```

## Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |

| | |
|---|---|
| | If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1). |
| | If an attempt is made to write a 1 to any reserved bit in CR4. |
| | If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If an attempt is made to write a 1 to any reserved bit in CR4. |
| | If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0). |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | These instructions cannot be executed in virtual-8086 mode. |

### Compatibility Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1). |
| | If an attempt is made to write a 1 to any reserved bit in CR3. |
| | If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5]. |
| #UD | If the LOCK prefix is used. |

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| | If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1). |
| | Attempting to clear CR0.PG[bit 32]. |
| | If an attempt is made to write a 1 to any reserved bit in CR4. |
| | If an attempt is made to write a 1 to any reserved bit in CR8. |
| | If an attempt is made to write a 1 to any reserved bit in CR3. |
| | If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5]. |
| #UD | If the LOCK prefix is used. |

# MOV—Move to/from Debug Registers

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 21/*r* | MOV *r32,* DR0-DR7 | N.E. | Valid | Move debug register to *r32* |
| 0F 21/*r* | MOV *r64,* DR0-DR7 | Valid | N.E. | Move extended debug register to *r64*. |
| 0F 23 /*r* | MOV DR0-DR7, *r32* | N.E. | Valid | Move *r32* to debug register |
| 0F 23 /*r* | MOV DR0-DR7, *r64* | Valid | N.E. | Move *r64* to extended debug register. |

## Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Chapter 18, “Debugging and Performance Monitoring”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE flag in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the IA-32 Architecture beginning with the Pentium processor.)

At the opcode level, the *reg* field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the *mod* field are always 11. The *r/m* field specifies the general-purpose register loaded or read.

In 64-bit mode, the instruction’s default operation size is 64 bits. Use of the REX.B prefix permits access to additional registers (R8-R15). Use of the REX.W or 66H prefix is ignored. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF ((DE = 1)  and (SRC or DEST = DR4 or DR5))
    THEN
        #UD;
    ELSE
        DEST ←SRC;

FI;
```

## Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| #UD | If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. |
| | If the LOCK prefix is used. |
| #DB | If any debug register is accessed while the DR7.GD[bit 13] = 1. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #UD | If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. |
| | If the LOCK prefix is used. |
| #DB | If any debug register is accessed while the DR7.GD[bit 13] = 1. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | The debug registers cannot be loaded or read when in virtual-8086 mode. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the current privilege level is not 0. |
| #UD | If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. |
| | If the LOCK prefix is used. |
| #DB | If any debug register is accessed while the DR7.GD[bit 13] = 1. |

# MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 28 /r | MOVAPD *xmm1, xmm2/m128* | Valid | Valid | Move packed double-precision floating-point values from *xmm2/m128* to *xmm1*. |
| 66 0F 29 /r | MOVAPD *xmm2/m128, xmm1* | Valid | Valid | Move packed double-precision floating-point values from *xmm1* to *xmm2/m128*. |

## Description

Moves a double quadword containing two packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

To move double-precision floating-point values to and from unaligned memory locations, use the MOVUPD instruction.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST ←SRC;
(* #GP if SRC or DEST unaligned memory operand *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_load_pd(double * p)

void _mm_store_pd(double *p, __m128 a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 28 /r | MOVAPS *xmm1, xmm2/m128* | Valid | Valid | Move packed single-precision floating-point values from *xmm2/m128* to *xmm1*. |
| 0F 29 /r | MOVAPS *xmm2/m128, xmm1* | Valid | Valid | Move packed single-precision floating-point values from *xmm1* to *xmm2/m128*. |

## Description

Moves a double quadword containing four packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) is generated.

To move packed single-precision floating-point values to or from unaligned memory locations, use the MOVUPS instruction.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST ←SRC;
(* #GP if SRC or DEST unaligned memory operand *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_load_ps (float * p)

void _mm_store_ps (float *p, __m128 a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

# MOVD/MOVQ—Move Doubleword/Move Quadword

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 6E /*r* | MOVD *mm, r/m32* | Valid | Valid | Move doubleword from *r/m32* to *mm*. |
| REX.W + 0F 6E /*r* | MOVQ *mm, r/m64* | Valid | N.E. | Move quadword from *r/m64* to *mm*. |
| 0F 7E /*r* | MOVD *r/m32, mm* | Valid | Valid | Move doubleword from *mm* to *r/m32*. |
| REX.W + 0F 7E /*r* | MOVQ *r/m64, mm* | Valid | N.E. | Move quadword from *mm* to *r/m64*. |
| 66 0F 6E /*r* | MOVD *xmm, r/m32* | Valid | Valid | Move doubleword from *r/m32* to *xmm*. |
| 66 REX.W 0F 6E /*r* | MOVQ *xmm, r/m64* | Valid | N.E. | Move quadword from *r/m64* to *xmm*. |
| 66 0F 7E /*r* | MOVD *r/m32, xmm* | Valid | Valid | Move doubleword from *xmm* register to *r/m32*. |
| 66 REX.W 0F 7E /*r* | MOVQ *r/m64, xmm* | Valid | N.E. | Move quadword from *xmm* register to *r/m64*. |

## Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

MOVD instruction when destination operand is MMX technology register:

```
DEST[31:0] ←SRC;
DEST[63:32] ←00000000H;
```

MOVD instruction when destination operand is XMM register:

```
DEST[31:0] ←SRC;
DEST[127:32] ←000000000000000000000000H;
```

MOVD instruction when source operand is MMX technology or XMM register:

```
DEST ←SRC[31:0];
```

MOVQ instruction when destination operand is XMM register:

```
DEST[63:0] ←SRC[63:0];
DEST[127:64] ←0000000000000000H;
```

MOVQ instruction when destination operand is r/m64:

```
DEST[63:0] ←SRC[63:0];
```

MOVQ instruction when source operand is XMM register or r/m64:

```
DEST ←SRC[63:0];
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVD __m64 _mm_cvtsi32_si64 (int i )

MOVD int _mm_cvtsi64_si32 ( __m64m )

MOVD __m128i _mm_cvtsi32_si128 (int a)

MOVD int _mm_cvtsi128_si32 ( __m128i a)

## Flags Affected

None.

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination operand is in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br><br>128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.<br><br>If the LOCK prefix is used. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | (MMX register operations only) If there is a pending FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If any part of the operand lies outside of the effective address space from 0 to FFFFH. |
| #UD | If CR0.EM[bit 2] = 1.<br><br>128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.<br><br>If the LOCK prefix is used. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | (MMX register operations only) If there is a pending FPU exception. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>(XMM register operations only) if CR4.OSFXSR[bit 9] = 0.<br>(XMM register operations only) if CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | (MMX register operations only) If there is a pending FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVDDUP—Move One Double-FP and Duplicate

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 12 /r | MOVDDUP *xmm1, xmm2/m64* | Valid | Valid | Move one double-precision floating-point value from the lower 64-bit operand in *xmm2/m64* to *xmm1* and duplicate. |

## Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 8 bytes of data at memory location m64 are loaded. When the register-register form of this operation is used, the lower half of the 128-bit source register is duplicated and copied into the 128-bit destination register. See Figure 3-14.

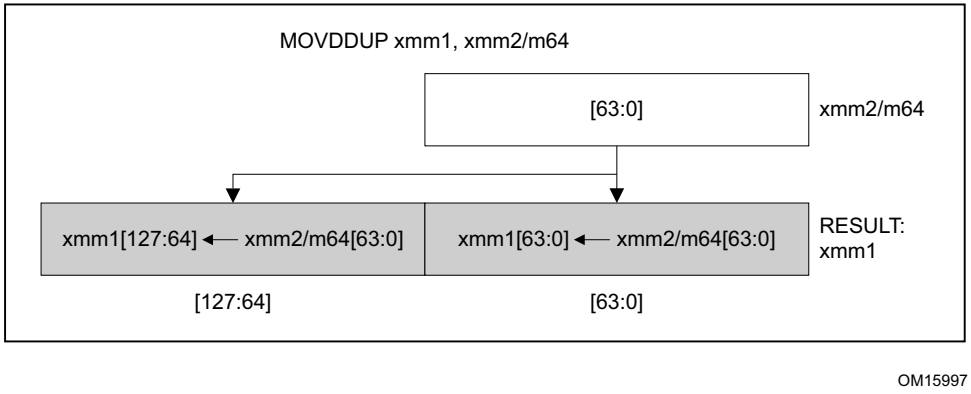



**Figure 3-14. MOVDDUP—Move One Double-FP and Duplicate**

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
IF (Source == m64)
    THEN
        (* Load instruction *)
        xmm1[63:0] = m64;
```

```
        xmm1[127:64] = m64;
    ELSE
        (* Move instruction *)
        xmm1[63:0] = xmm2[63:0];
        xmm1[127:64] = xmm2[63:0];
FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVDDUP __m128d _mm_movedup_pd(__m128d a)

__m128d _mm_loaddup_pd(double const * dp)

## Exceptions

None

## Numeric Exceptions

None

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.SSE3(ECX, bit 0) is 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVDQA—Move Aligned Double Quadword

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 6F /r | MOVDQA *xmm1, xmm2/m128* | Valid | Valid | Move aligned double quadword from *xmm2/m128* to *xmm1*. |
| 66 0F 7F /r | MOVDQA *xmm2/m128, xmm1* | Valid | Valid | Move aligned double quadword from *xmm1* to *xmm2/m128*. |

## Description

Moves a double quadword from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

To move a double quadword to or from unaligned memory locations, use the MOVDQU instruction.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST ←SRC;
(* #GP if SRC or DEST unaligned memory operand *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVDQA __m128i _mm_load_si128 ( __m128i *p)

MOVDQA void _mm_store_si128 ( __m128i *p, __m128i a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #PF(fault-code) | If a page fault occurs. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |

| | |
|---|---|
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside of the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

# MOVDQU—Move Unaligned Double Quadword

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 6F /r | MOVDQU *xmm1, xmm2/m128* | Valid | Valid | Move unaligned double quadword from *xmm2/m128* to *xmm1*. |
| F3 0F 7F /r | MOVDQU *xmm2/m128, xmm1* | Valid | Valid | Move unaligned double quadword from *xmm1* to *xmm2/m128*. |

## Description

Moves a double quadword from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.

To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the MOVDQA instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

DEST ←SRC;

## Intel C/C++ Compiler Intrinsic Equivalent

MOVDQU void _mm_storeu_si128 ( __m128i *p, __m128i a)

MOVDQU __m128i _mm_loadu_si128 ( __m128i *p)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| #PF(fault-code) | If a page fault occurs. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If any part of the operand lies outside of the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# MOVDQ2Q—Move Quadword from XMM to MMX Technology Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F D6 | MOVDQ2Q *mm, xmm* | Valid | Valid | Move low quadword from *xmm* to *mmx* register. |

## Description

Moves the low quadword from the source operand (second operand) to the destination operand (first operand). The source operand is an XMM register and the destination operand is an MMX technology register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVDQ2Q instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST ←SRC[63:0];
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVDQ2Q __m64 _mm_movepi64_pi64 ( __m128i a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #MF | If there is a pending x87 FPU exception. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# MOVHLPS— Move Packed Single-Precision Floating-Point Values High to Low

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 12 /r | MOVHLPS *xmm1, xmm2* | Valid | Valid | Move two packed single-precision floating-point values from high quadword of *xmm2* to low quadword of *xmm1*. |

## Description

Moves two packed single-precision floating-point values from the high quadword of the source operand (second operand) to the low quadword of the destination operand (first operand). The high quadword of the destination operand is left unchanged.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←SRC[127:64];
(* DEST[127:64] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Real Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual 8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# MOVHPD—Move High Packed Double-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 16 /r | MOVHPD *xmm, m64* | Valid | Valid | Move double-precision floating-point value from *m64* to high quadword of *xmm*. |
| 66 0F 17 /r | MOVHPD *m64, xmm* | Valid | Valid | Move double-precision floating-point value from high quadword of *xmm* to *m64*. |

## Description

Moves a double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory location. This instruction allows a double-precision floating-point value to be moved to and from the high quadword of an XMM register and memory. It cannot be used for register to register or memory to memory moves. When the destination operand is an XMM register, the low quadword of the register remains unchanged.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

MOVHPD instruction for memory to XMM move:

```
DEST[127:64] ←SRC;
(* DEST[63:0] unchanged *)
```

MOVHPD instruction for XMM to memory move:

```
DEST ←SRC[127:64];
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD __m128d _mm_loadh_pd ( __m128d a, double *p)

MOVHPD void _mm_storeh_pd (double *p, __m128d a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |

| | |
|---|---|
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVHPS—Move High Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 16 /r | MOVHPS *xmm, m64* | Valid | Valid | Move two packed single-precision floating-point values from *m64* to high quadword of *xmm*. |
| 0F 17 /r | MOVHPS *m64, xmm* | Valid | Valid | Move two packed single-precision floating-point values from high quadword of *xmm* to *m64*. |

## Description

Moves two packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory location. This instruction allows two single-precision floating-point values to be moved to and from the high quadword of an XMM register and memory. It cannot be used for register to register or memory to memory moves. When the destination operand is an XMM register, the low quadword of the register remains unchanged.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

MOVHPS instruction for memory to XMM move:

```
DEST[127:64] ←SRC;
(* DEST[63:0] unchanged *)
```

MOVHPS instruction for XMM to memory move:

```
DEST ←SRC[127:64];
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS __m128d _mm_loadh_pi ( __m128d a, __m64 *p)

MOVHPS void _mm_storeh_pi (__m64 *p, __m128d a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |

| | |
|---|---|
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| OF 16 /r | MOVLHPS *xmm1, xmm2* | Valid | Valid | Move two packed single-precision floating-point values from low quadword of *xmm2* to high quadword of *xmm1*. |

## Description

Moves two packed single-precision floating-point values from the low quadword of the source operand (second operand) to the high quadword of the destination operand (first operand). The low quadword of the destination operand is left unchanged.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[127:64] ←SRC[63:0];
(* DEST[63:0] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Real Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual 8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# MOVLPD—Move Low Packed Double-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 12 /r | MOVLPD *xmm, m64* | Valid | Valid | Move double-precision floating-point value from *m64* to low quadword of *xmm* register. |
| 66 0F 13 /r | MOVLPD *m64, xmm* | Valid | Valid | Move double-precision floating-point nvalue from low quadword of *xmm* register to *m64*. |

## Description

Moves a double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory location. This instruction allows a double-precision floating-point value to be moved to and from the low quadword of an XMM register and memory. It cannot be used for register to register or memory to memory moves. When the destination operand is an XMM register, the high quadword of the register remains unchanged.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

MOVLPD instruction for memory to XMM move:

```
DEST[63:0] ←SRC;
(* DEST[127:64] unchanged *)
```

MOVLPD instruction for XMM to memory move:

```
DEST ←SRC[63:0];
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD __m128d _mm_loadl_pd ( __m128d a, double *p)

MOVLPD void _mm_storel_pd (double *p, __m128d a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |

| | |
|---|---|
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVLPS—Move Low Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 12 /r | MOVLPS *xmm, m64* | Valid | Valid | Move two packed single-precision floating-point values from *m64* to low quadword of *xmm*. |
| 0F 13 /r | MOVLPS *m64, xmm* | Valid | Valid | Move two packed single-precision floating-point values from low quadword of *xmm* to *m64*. |

## Description

Moves two packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). The source and destination operands can be an XMM register or a 64-bit memory location. This instruction allows two single-precision floating-point values to be moved to and from the low quadword of an XMM register and memory. It cannot be used for register to register or memory to memory moves. When the destination operand is an XMM register, the high quadword of the register remains unchanged.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

MOVLPD instruction for memory to XMM move:

```
DEST[63:0] ←SRC;
(* DEST[127:64] unchanged *)
```

MOVLPD instruction for XMM to memory move:

```
DEST ←SRC[63:0];
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS __m128 _mm_loadl_pi ( __m128 a, __m64 *p)

MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |

| | |
|---|---|
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 50 /r | MOVMSKPD *r32, xmm* | Valid | Valid | Extract 2-bit sign mask from *xmm* and store in *r32*. |
| 66 REX.W 0F 50 /r | MOVMSKPD *r64, xmm* | Valid | N.E. | Extract 2-bit sign mask from *xmm* and store in *r64*. Zero extend 32-bit results to 64-bits. |

## Description

Extracts the sign bits from the packed double-precision floating-point values in the source operand (second operand), formats them into a 2-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
DEST[0] ←SRC[63];
DEST[1] ←SRC[127];

IF DEST = r32
    THEN DEST[31:2] ←ZeroExtend;
    ELSE DEST[63:2] ←ZeroExtend;
FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVMSKPD int _mm_movemask_pd ( __m128 a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 50 /r | MOVMSKPS *r32, xmm* | Valid | Valid | Extract 4-bit sign mask from *xmm* and store in *r32*. |
| REX.W + 0F 50 /r | MOVMSKPS *r64, xmm* | Valid | N.E. | Extract 4-bit sign mask from *xmm* and store in *r64*. Zero extend 32-bit results to 64-bits. |

## Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 low-order bits of the destination operand.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. Use of the REX.W prefix promotes the instruction to 64-bit operands. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
DEST[0] ←SRC[31];
DEST[1] ←SRC[63];
DEST[2] ←SRC[95];
DEST[3] ←SRC[127];

IF DEST = r32
    THEN DEST[31:4] ←ZeroExtend;
    ELSE DEST[63:4] ←ZeroExtend;
FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

int _mm_movemask_ps(__m128 a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual 8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# MOVNTDQ—Store Double Quadword Using Non-Temporal Hint

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F E7 /r | MOVNTDQ *m128,* xmm | Valid | Valid | Move double quadword from *xmm* to *m128* using non-temporal hint. |

## Description

Moves the double quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, which is assumed to contain integer data (packed bytes, words, doublewords, or quadwords). The destination operand is a 128-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

DEST ←SRC;

## Intel C/C++Compiler Intrinsic Equivalent

MOVNTDQ void _mm_stream_pd( double* p, __m128d a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# MOVNTI—Store Doubleword Using Non-Temporal Hint

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F C3 /r | MOVNTI *m32, r32* | Valid | Valid | Move doubleword from *r32* to *m32* using non-temporal hint. |
| REX.W + 0F C3 /r | MOVNTI *m64, r64* | Valid | N.E. | Move quadword from *r64* to *m64* using non-temporal hint. |

## Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
DEST ←SRC;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVNTI  void _mm_stream_si32 (int *p, int a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #UD | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #UD | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #UD | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 2B /r | MOVNTPD *m128, xmm* | Valid | Valid | Move packed double-precision floating-point values from *xmm* to *m128* using non-temporal hint. |

## Description

Moves the double quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an XMM register, which is assumed to contain two packed double-precision floating-point values. The destination operand is a 128-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

DEST ←SRC;

## Intel C/C++ Compiler Intrinsic Equivalent

MOVNTPD void _mm_stream_pd(double *p, __m128d a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 2B /r | MOVNTPS *m128, xmm* | Valid | Valid | Move packed single-precision floating-point values from *xmm* to *m128* using non-temporal hint. |

## Description

Moves the double quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an XMM register, which is assumed to contain four packed single-precision floating-point values. The destination operand is a 128-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST ←SRC;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVNTDQ void _mm_stream_ps(float * p, __m128 a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

# MOVNTQ—Store of Quadword Using Non-Temporal Hint

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F E7 /r | MOVNTQ *m64, mm* | Valid | Valid | Move quadword from *mm* to *m64* using non-temporal hint. |

## Description

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## Operation

```
DEST ←SRC;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVNTQ  void _mm_stream_pi(__m64 * p, __m64 a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |

| | |
|---|---|
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | If there is a pending x87 FPU exception. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVQ—Move Quadword

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 6F /*r* | MOVQ *mm, mm/m64* | Valid | Valid | Move quadword from *mm/m64* to *mm*. |
| 0F 7F /*r* | MOVQ *mm/m64, mm* | Valid | Valid | Move quadword from *mm* to *mm/m64*. |
| F3 0F 7E | MOVQ *xmm1, xmm2/m64* | Valid | Valid | Move quadword from *xmm2/mem64* to *xmm1*. |
| 66 0F D6 | MOVQ *xmm2/m64, xmm1* | Valid | Valid | Move quadword from *xmm1* to *xmm2/mem64*. |

## Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

MOVQ instruction when operating on MMX technology registers and memory locations:

```
DEST ←SRC;
```

MOVQ instruction when source and destination operands are XMM registers:

```
DEST[63:0] ←SRC[63:0];
```

MOVQ instruction when source operand is XMM register and destination operand is memory location:

```
DEST ←SRC[63:0];
```

MOVQ instruction when source operand is memory location and destination operand is XMM register:

```
DEST[63:0] ←SRC;
```

```
DEST[127:64] ←0000000000000000H;
```

## Flags Affected

None.

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination operand is in a non-writable segment.<br>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #UD | If CR0.EM[bit 2] = 1.<br>128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.<br>If the LOCK prefix is used. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | (MMX register operations only) If there is a pending FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If any part of the operand lies outside of the effective address space from 0 to FFFFH. |
| #UD | If CR0.EM[bit 2] = 1.<br>128-bit operations will generate #UD only if CR4.OSFXSR[bit 9] = 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.<br>If the LOCK prefix is used. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | (MMX register operations only) If there is a pending FPU exception. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #UD | If CR0.EM[bit 2] = 1. |
| | (XMM register operations only) If CR4.OSFXSR[bit 9] = 0. |
| | (XMM register operations only) If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #NM | If CR0.TS[bit 3] = 1. |
| #MF | (MMX register operations only) If there is a pending FPU exception. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F D6 | MOVQ2DQ *xmm, mm* | Valid | Valid | Move quadword from *mmx* to low quadword of *xmm*. |

## Description

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ2DQ instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←SRC[63:0];
DEST[127:64] ←0000000000000000H;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVQ2DQ __128i _mm_movpi64_pi64 ( __m64 a)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #MF | If there is a pending x87 FPU exception. |

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

# MOVS/MOVSB/MOVSW/MOVSD/MOVSQ—Move Data from String to String

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| A4 | MOVS *m8, m8* | Valid | Valid | For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R\|E)SI to (R\|E)DI. |
| A5 | MOVS *m16, m16* | Valid | Valid | For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R\|E)SI to (R\|E)DI. |
| A5 | MOVS *m32, m32* | Valid | Valid | For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R\|E)SI to (R\|E)DI. |
| REX.W + A5 | MOVS *m64, m64* | Valid | N.E. | Move qword from address (R\|E)SI to (R\|E)DI. |
| A4 | MOVSB | Valid | Valid | For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R\|E)SI to (R\|E)DI. |
| A5 | MOVSW | Valid | Valid | For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R\|E)SI to (R\|E)DI. |
| A5 | MOVSD | Valid | Valid | For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R\|E)SI to (R\|E)DI. |
| REX.W + A5 | MOVSQ | Valid | N.E. | Move qword from address (R\|E)SI to (R\|E)DI. |

## Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). Both the source and destination operands are located in memory. The address of the source operand is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the destination operand is read from the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the MOVS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source and destination operands should be symbols that indicate the size and location of the source value and the destination, respectively. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source and destination operand symbols must specify the correct **type** (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct **location**. The locations of the source and destination operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the move string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the MOVS instructions. Here also DS:(E)SI and ES:(E)DI are assumed to be the source and destination operands, respectively. The size of the source and destination operands is selected with the mnemonic: MOVSB (byte move), MOVSW (word move), or MOVSD (doubleword move).

After the move operation, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incremented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The MOVS, MOVSB, MOVSW, and MOVSD instructions can be preceded by the REP prefix (see "REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*) for block moves of ECX bytes, words, or doublewords.

In 64-bit mode, the instruction's default address size is 64 bits, 32-bit address size is supported using the prefix 67H. The 64-bit addresses are specified by RSI and RDI; 32-bit address are specified by ESI and EDI. Use of the REX.W prefix promotes doubleword operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
DEST ←SRC;

Non-64-bit Mode:

IF (Byte move)
    THEN IF DF = 0
        THEN
            (E)SI ←(E)SI +1;
            (E)DI ←(E)DI +1;
        ELSE
```

```
            (E)SI ←(E)SI – 1;
            (E)DI ←(E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (E)SI ←(E)SI +2;
            (E)DI ←(E)DI +2;
            FI;
        ELSE
            (E)SI ←(E)SI – 2;
            (E)DI ←(E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (E)SI ←(E)SI +4;
            (E)DI ←(E)DI +4;
            FI;
        ELSE
            (E)SI ←(E)SI – 4;
            (E)DI ←(E)DI – 4;
        FI;
FI;
64-bit Mode:

IF (Byte move)
    THEN IF DF = 0
        THEN
            (R|E)SI ←(R|E)SI +1;
            (R|E)DI ←(R|E)DI +1;
        ELSE
            (R|E)SI ←(R|E)SI – 1;
            (R|E)DI ←(R|E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R|E)SI ←(R|E)SI +2;
            (R|E)DI ←(R|E)DI +2;
            FI;
        ELSE
            (R|E)SI ←(R|E)SI – 2;
            (R|E)DI ←(R|E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
```

```
            (R|E)SI ←(R|E)SI +4;
            (R|E)DI ←(R|E)DI +4;
            FI;
        ELSE
            (R|E)SI ←(R|E)SI – 4;
            (R|E)DI ←(R|E)DI – 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R|E)SI ←(R|E)SI +8;
            (R|E)DI ←(R|E)DI +8;
            FI;
        ELSE
            (R|E)SI ←(R|E)SI – 8;
            (R|E)DI ←(R|E)DI – 8;
        FI;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a non-writable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# MOVSD—Move Scalar Double-Precision Floating-Point Value

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 10 /r | MOVSD *xmm1, xmm2/m64* | Valid | Valid | Move scalar double-precision floating-point value from *xmm2/m64* to *xmm1* register. |
| F2 0F 11 /r | MOVSD *xmm2/m64, xmm1* | Valid | Valid | Move scalar double-precision floating-point value from *xmm1* register to *xmm2/m64*. |

## Description

Moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

When the source and destination operands are XMM registers, the high quadword of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the high quadword of the destination operand is cleared to all 0s.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

MOVSD instruction when source and destination operands are XMM registers:

```
DEST[63:0] ←SRC[63:0];
(* DEST[127:64] unchanged *)
```

MOVSD instruction when source operand is XMM register and destination operand is memory location:

```
DEST ←SRC[63:0];
```

MOVSD instruction when source operand is memory location and destination operand is XMM register:

```
DEST[63:0] ←SRC;
DEST[127:64] ←0000000000000000H;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVSD __m128d _mm_load_sd (double *p)

MOVSD void _mm_store_sd (double *p, __m128d a)

MOVSD __m128d _mm_store_sd (__m128d a, __m128d b)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVSHDUP—Move Packed Single-FP High and Duplicate

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 16 /r | MOVSHDUP *xmm1, xmm2/m128* | Valid | Valid | Move two single-precision floating-point values from the higher 32-bit operand of each qword in *xmm2/m128* to *xmm1* and duplicate each 32-bit operand to the lower 32-bits of each qword. |

## Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded and the single-precision elements in positions 1 and 3 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register. See Figure 3-15.

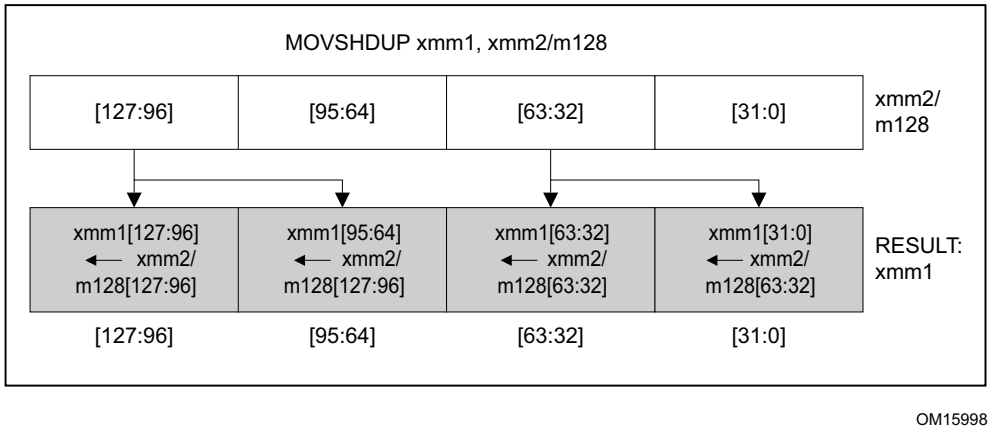



**Figure 3-15. MOVSHDUP—Move Packed Single-FP High and Duplicate**

In 64-bit mode, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
IF (Source == m128)
    THEN     (* Load instruction *)
        xmm1[31:0] = m128[63:32];
        xmm1[63:32] = m128[63:32];
        xmm1[95:64]  = m128[127:96];
        xmm1[127:96] = m128[127:96];
    ELSE     (* Move instruction *)
        xmm1[31:0] = xmm2[63:32];
        xmm1[63:32] = xmm2[63:32];
        xmm1[95:64] = xmm2[127:96];
        xmm1[127:96] = xmm2[127:96];
FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVSHDUP __m128 _mm_movehdup_ps(__m128 a)

## Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

## Numeric Exceptions

None

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is not non-canonical. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.SSE3(ECX, bit 0) is 0. |
| | If the LOCK prefix is used. |

# MOVSLDUP—Move Packed Single-FP Low and Duplicate

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 12 /r | MOVSLDUP *xmm1, xmm2/m128* | Valid | Valid | Move two single-precision floating-point values from the lower 32-bit operand of each qword in *xmm2/m128* to *xmm1* and duplicate each 32-bit operand to the higher 32-bits of each qword. |

## Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded and the single-precision elements in positions 0 and 2 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register.

See Figure 3-16.

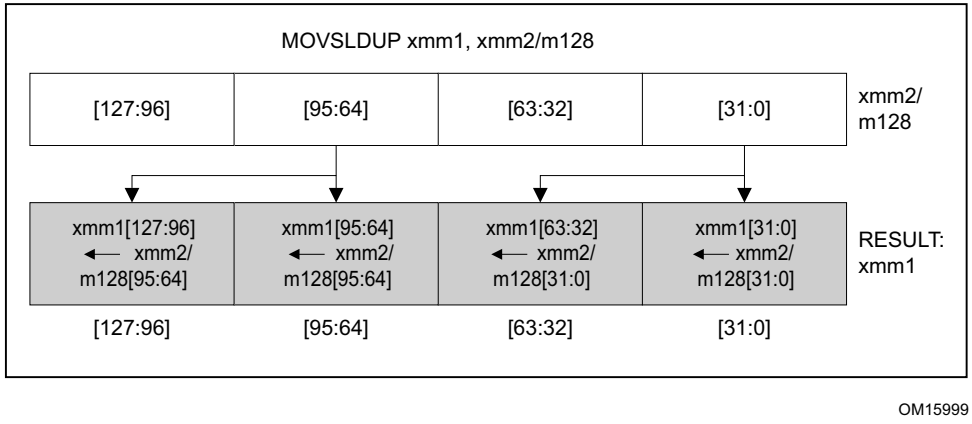



Figure 3-16. MOVSLDUP—Move Packed Single-FP Low and Duplicate

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
IF (Source == m128)
    THEN     (* Load instruction *)
        xmm1[31:0] = m128[31:0];
        xmm1[63:32] = m128[31:0];
        xmm1[95:64] = m128[95:64];
        xmm1[127:96] = m128[95::64];
    ELSE     (* Move instruction *)
        xmm1[31:0] = xmm2[31:0];
        xmm1[63:32] = xmm2[31:0];
        xmm1[95:64] = xmm2[95:64];
        xmm1[127:96] = xmm2[95:64];
FI;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVSLDUP __m128 _mm_moveldup_ps(__m128 a)

## Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

## Numeric Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Real Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1.If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |

### Virtual 8086 Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:ECX.SSE3[bit 0] = 0. |
| | If the LOCK prefix is used. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.SSE3(ECX, bit 0) is 0. |
| | If the LOCK prefix is used. |

# MOVSS—Move Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 10 /r | MOVSS *xmm1, xmm2/m32* | Valid | Valid | Move scalar single-precision floating-point value from *xmm2/m32* to *xmm1* register. |
| F3 0F 11 /r | MOVSS *xmm2/m32, xmm* | Valid | Valid | Move scalar single-precision floating-point value from *xmm1* register to *xmm2/m32*. |

## Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

When the source and destination operands are XMM registers, the three high-order doublewords of the destination operand remain unchanged. When the source operand is a memory location and destination operand is an XMM registers, the three high-order doublewords of the destination operand are cleared to all 0s.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

MOVSS instruction when source and destination operands are XMM registers:

```
DEST[31:0] ←SRC[31:0];
(* DEST[127:32] remains unchanged *)
```

MOVSS instruction when source operand is XMM register and destination operand is memory location:

```
DEST ←SRC[31:0];
```

MOVSS instruction when source operand is memory location and destination operand is XMM register:

```
DEST[31:0] ←SRC;
DEST[127:32] ←00000000000000000000000000H;
```

## Intel C/C++ Compiler Intrinsic Equivalent

MOVSS __m128 _mm_load_ss(float * p)

MOVSS void _mm_store_ss(float * p, __m128 a)

MOVSS __m128 _mm_move_ss(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

## Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MOVSX/MOVSXD—Move with Sign-Extension

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F BE /r | MOVSX *r16, r/m8* | Valid | Valid | Move byte to word with sign-extension. |
| 0F BE /r | MOVSX *r32, r/m8* | Valid | Valid | Move byte to doubleword with sign-extension. |
| REX + 0F BE /r | MOVSX *r64, r/m8** | Valid | N.E. | Move byte to quadword with sign-extension. |
| 0F BF /r | MOVSX *r32, r/m16* | Valid | Valid | Move word to doubleword, with sign-extension. |
| REX.W + 0F BF /r | MOVSX *r64, r/m16* | Valid | N.E. | Move word to quadword with sign-extension. |
| REX.W** + 63 /r | MOVSXD *r64, r/m32* | Valid | N.E. | Move doubleword to quadword with sign-extension. |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** The use of MOVSXD without REX.W in 64-bit mode is discouraged, Regular MOV should be used instead of using MOVSXD without REX.W.

## Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
DEST ←SignExtend(SRC);
```

## Flags Affected

None.

### Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 10 /r | MOVUPD *xmm1, xmm2/m128* | Valid | Valid | Move packed double-precision floating-point values from *xmm2/m128* to xmm1. |
| 66 0F 11 /r | MOVUPD *xmm2/m128, xmm* | Valid | Valid | Move packed double-precision floating-point values from xmm1 to *xmm2/m128*. |

## Description

Moves a double quadword containing two packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers. When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.

To move double-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPD instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST ←SRC;
```

## Intel C/C++Compiler Intrinsic Equivalent

MOVUPD __m128 _mm_loadu_pd(double * p)

MOVUPD void _mm_storeu_pd(double *p, __m128 a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 10 /r | MOVUPS *xmm1, xmm2/m128* | Valid | Valid | Move packed single-precision floating-point values from *xmm2/m128* to xmm1. |
| 0F 11 /r | MOVUPS *xmm2/m128, xmm1* | Valid | Valid | Move packed single-precision floating-point values from xmm1 to *xmm2/m128*. |

## Description

Moves a double quadword containing four packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers. When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.

To move packed single-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPS instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

DEST ←SRC;

## Intel C/C++ Compiler Intrinsic Equivalent

MOVUPS __m128 _mm_loadu_ps(double * p)

MOVUPS void _mm_storeu_ps(double *p, __m128 a)

## SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #UD | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #UD | If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

# MOVZX—Move with Zero-Extend

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F B6 /*r* | MOVZX *r16, r/m8* | Valid | Valid | Move byte to word with zero-extension. |
| 0F B6 /*r* | MOVZX *r32, r/m8* | Valid | Valid | Move byte to doubleword, zero-extension. |
| REX.W + 0F B6 /*r* | MOVZX *r64, r/m8** | Valid | N.E. | Move byte to quadword, zero-extension. |
| 0F B7 /*r* | MOVZX *r32, r/m16* | Valid | Valid | Move word to doubleword, zero-extension. |
| REX.W + 0F B7 /*r* | MOVZX *r64, r/m16* | Valid | N.E. | Move word to quadword, zero-extension. |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if the REX prefix is used: AH, BH, CH, DH.

## Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
DEST ←ZeroExtend(SRC);
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.<br>If the DS, ES, FS, or GS register contains a NULL segment selector. |

| | |
|---|---|
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

# MUL—Unsigned Multiply

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F6 /4 | MUL *r/m8* | Valid | Valid | Unsigned multiply (AX ←AL ∗ *r/m8*). |
| REX + F6 /4 | MUL *r/m8** | Valid | N.E. | Unsigned multiply (AX ←AL ∗ *r/m8*). |
| F7 /4 | MUL *r/m16* | Valid | Valid | Unsigned multiply (DX:AX ←AX ∗ *r/m16*). |
| F7 /4 | MUL *r/m32* | Valid | Valid | Unsigned multiply (EDX:EAX ←EAX ∗ *r/m32*). |
| REX.W + F7 /4 | MUL *r/m64* | Valid | N.E. | Unsigned multiply (RDX:RAX ←RAX ∗ *r/m64.* |

**NOTES:**

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 3-61.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

**Table 3-61. MUL Results**

| Operand Size | Source 1 | Source 2 | Destination |
|---|---|---|---|
| Byte | AL | r/m8 | AX |
| Word | AX | r/m16 | DX:AX |
| Doubleword | EAX | r/m32 | EDX:EAX |
| Quadword | RAX | r/m64 | RDX:RAX |

## Operation

```
IF (Byte operation)
    THEN
        AX ←AL * SRC;
    ELSE (* Word or doubleword operation *)
        IF OperandSize = 16
            THEN
                DX:AX ←AX * SRC;
            ELSE IF OperandSize = 32
                THEN EDX:EAX ←EAX * SRC; FI;
            ELSE (* OperandSize = 64 *)
                RDX:RAX ←RAX * SRC;
        FI;
FI;
```

## Flags Affected

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a NULL segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |
| #UD | If the LOCK prefix is used. |

## Real-Address Mode Exceptions

| | |
|---|---|
| #GP | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| #SS | If a memory operand effective address is outside the SS segment limit. |
| #UD | If the LOCK prefix is used. |

## Virtual-8086 Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |

| | |
|---|---|
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made. |
| #UD | If the LOCK prefix is used. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MULPD—Multiply Packed Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 66 0F 59 /r | MULPD *xmm1, xmm2/m128* | Valid | Valid | Multiply packed double-precision floating-point values in *xmm2/m128* by *xmm1*. |

## Description

Performs a SIMD multiply of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←DEST[63:0] * SRC[63:0];
DEST[127:64] ←DEST[127:64] * SRC[127:64];
```

## Intel C/C++ Compiler Intrinsic Equivalent

MULPD __m128d _mm_mul_pd (m128d a, m128d b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form.<br>If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

# MULPS—Multiply Packed Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 59 /r | MULPS *xmm1, xmm2/m128* | Valid | Valid | Multiply packed single-precision floating-point values in *xmm2/mem* by *xmm1*. |

## Description

Performs a SIMD multiply of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←DEST[31:0] * SRC[31:0];
DEST[63:32] ←DEST[63:32] * SRC[63:32];
DEST[95:64] ←DEST[95:64] * SRC[95:64];
DEST[127:96] ←DEST[127:96] * SRC[127:96];
```

## Intel C/C++ Compiler Intrinsic Equivalent

MULPS __m128 _mm_mul_ps(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.<br>If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Real-Address Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| | If memory operand is not aligned on a 16-byte boundary, regardless of segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |

| | |
|---|---|
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

# MULSD—Multiply Scalar Double-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F2 0F 59 /r | MULSD *xmm1, xmm2/m64* | Valid | Valid | Multiply the low double-precision floating-point value in *xmm2/mem64* by low double-precision floating-point value in *xmm1*. |

## Description

Multiplies the low double-precision floating-point value in the source operand (second operand) by the low double-precision floating-point value in the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. The high quadword of the destination operand remains unchanged. See Figure 11-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[63:0] ←DEST[63:0] * xmm2/m64[63:0];
(* DEST[127:64] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

MULSD __m128d _mm_mul_sd (m128d a, m128d b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE2[bit 26] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC | For unaligned memory reference. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE2[bit 26] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MULSS—Multiply Scalar Single-Precision Floating-Point Values

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| F3 0F 59 /r | MULSS *xmm1, xmm2/m32* | Valid | Valid | Multiply the low single-precision floating-point value in *xmm2/mem* by the low single-precision floating-point value in *xmm1*. |

## Description

Multiplies the low single-precision floating-point value from the source operand (second operand) by the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order double-words of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

## Operation

```
DEST[31:0] ←DEST[31:0] * SRC[31:0];
(* DEST[127:32] unchanged *)
```

## Intel C/C++ Compiler Intrinsic Equivalent

MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. |
| #SS(0) | For an illegal address in the SS segment. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

### Real-Address Mode Exceptions

| | |
|---|---|
| GP(0) | If any part of the operand lies outside the effective address space from 0 to FFFFH. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.<br>If CR0.EM[bit 2] = 1.<br>If CR4.OSFXSR[bit 9] = 0.<br>If CPUID.01H:EDX.SSE[bit 25] = 0.<br>If the LOCK prefix is used. |

### Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

| | |
|---|---|
| #PF(fault-code) | For a page fault. |
| #AC | For unaligned memory reference. |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #SS(0) | If a memory address referencing the SS segment is in a non-canonical form. |
| #GP(0) | If the memory address is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #NM | If CR0.TS[bit 3] = 1. |
| #XM | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1. |

| | |
|---|---|
| #UD | If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0. |
| | If CR0.EM[bit 2] = 1. |
| | If CR4.OSFXSR[bit 9] = 0. |
| | If CPUID.01H:EDX.SSE[bit 25] = 0. |
| | If the LOCK prefix is used. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# MWAIT—Monitor Wait

| Opcode | Instruction | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|
| 0F 01 *C9* | MWAIT | Valid | Valid | A hint that allow the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events. |

## Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

A CPUID feature flag (ECX bit 3; CPUID executed EAX = 1) indicates the availability of MONITOR and MWAIT in the processor. When set, the unconditional execution of MWAIT is supported at privilege levels 0; conditional execution is supported at privilege levels 1 through 3 (test for the appropriate support before unconditional use). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLES MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an illegal opcode exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

## MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. For Pentium 4 processors (CPUID signature family 15 and model 3), non-zero values for EAX and ECX are reserved.

A store to the address range armed by the MONITOR instruction, an interrupt, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, or the RESET# signal will exit the implementation-dependent-optimized state. Note that an interrupt will cause the processor to exit only if the state was entered with interrupts enabled.

If a store to the address range causes the processor to exit, execution will resume at the instruction following the MWAIT instruction. If an interrupt (including NMI) caused the processor to exit the implementation-dependent-optimized state, the

processor will exit the state and handle the interrupt. If an SMI caused the processor to exit the implementation-dependent-optimized state, execution will resume at the instruction following MWAIT after handling of the SMI. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction. There may also be other implementation-dependent events or time-outs that may take the processor out of the implementation-dependent-optimized state and resume execution at the instruction following the MWAIT.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.

### MWAIT for Power Management

MWAIT accepts a hint and optional extension to the processor that it can enter a specified target C state while waiting for an event or a store operation to the address range armed by MONITOR. Support for MWAIT extensions for power management is indicated by CPUID.05H.ECX[0] reporting 1.

EAX and ECX will be used to communicate the additional information to the MWAIT instruction, such as the kind of optimized state the processor should enter. ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. A given processor implementation may choose to ignore the hint and continue executing the next instruction. Future processor implementations may implement several optimized “waiting” states and will select among those states based on the hint argument.

Table 3-62 describes the meaning of ECX and EAX registers for MWAIT extensions.

**Table 3-62. MWAIT Extension Register (ECX)**

| Bits | Description |
|---|---|
| 0 | Treat Interrupt as break-event, even when interrupts are disabled (EFLAGS.IF=0) |
| 31: 1 | Reserved |

**Table 3-63. MWAIT Hints Register (EAX)**

| Bits | Description |
|---|---|
| 3 : 0 | Sub C-state within a C-state, indicated by bits [7:4] |
| 7 : 4 | Target C-state*<br>Value of 0 means C1; 1 means C2 and so on<br>Value of 01111B means C0<br><br>Note: Target C states for MWAIT extensions are processor-specific C-states, not ACPI C-states |
| 31: 8 | Reserved |

Note that if MWAIT is used to enter any of the C-states that are numerically higher than C1, a store to the address range armed by the MONITOR instruction will cause the processor to exit MWAIT only if the store was originated by other processor agents. A store from non-processor agent may not cause the processor to exit MWAIT in such cases

For additional details of MWAIT extensions, see Chapter 13, "Power and Thermal Management," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

## Operation

```
(* MWAIT takes the argument in EAX as a hint extension and is architected to take the argument in ECX as an instruction extension MWAIT EAX, ECX *)
{
WHILE (! ("Monitor Hardware is in armed state")) {
    implementation_dependent_optimized_state(EAX, ECX); }
Set the state of Monitor Hardware as triggered;
}
```

## Intel C/C++ Compiler Intrinsic Equivalent

MWAIT  void _mm_mwait(unsigned extensions, unsigned hints)

## Example

MONITOR/MWAIT instruction pair must be coded in the same loop because execution of the MWAIT instruction will trigger the monitor hardware. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. Setting up MONITOR without executing MWAIT has no adverse effects.

Typically the MONITOR/MWAIT pair is used in a sequence, such as:

```
EAX = Logical Address(Trigger)
ECX = 0 (*Hints *)
```

```
EDX = 0 (* Hints *)

IF ( !trigger_store_happened) {
    MONITOR EAX, ECX, EDX
    IF ( !trigger_store_happened ) {
        MWAIT EAX, ECX
    }
}
```

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the monitor instruction. Without the second check that triggering store would go un-noticed. Typical usage of MONITOR and MWAIT would have the above code sequence within a loop.

## Numeric Exceptions

None

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. |
| | If ECX = 0. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | For a page fault. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |
| | If current privilege level is not 0. |

## Real Address Mode Exceptions

| | |
|---|---|
| #GP | If any part of the operand in the CS, DS, ES, FS, or GS segment lies outside of the effective address space from 0 to FFFFH. |
| | If ECX ≠ 0. |
| #SS | If any part of the operand in the SS segment lies outside of the effective address space from 0 to FFFFH. |
| #UD | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |

## Virtual 8086 Mode Exceptions

| | |
|---|---|
| #UD | The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1). |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

| | |
|---|---|
| #GP(0) | If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form. |
| | If RCX ≠ 0. |
| #SS(0) | If the linear address of the operand in the SS segment is in a non-canonical form. |
| #PF(fault-code) | For a page fault. |
| #UD | If the current privilege level is not 0. |
| | If CPUID.01H:ECX.MONITOR[bit 3] = 0. |